mPlane

an Intelligent Measurement Plane for Future Network and Application Management

ICT FP7-318627

mPlane Architecture Specification

Author(s):	POLITO	M. Mellia, A. Finamore
	SSB	S. Pentassuglia, G. De Rosa
	TI	F. Invernizzi
	EURECOM	M. Milanesio
	ENST	D. Rossi
	NEC	S. Niccolini
	TID	I. Leontiadis
	NETVISOR	T. Szemethy, B. Szabó
	FHA	R. Winter, M. Faath
	ULG	B. Donnet
	ETH	B. Trammell (ed.), M. Kühlewind
	A-LBELL	D. Papadimitriou

Document Number:	D1.4
Revision:	1.1
Revision Date:	15 Apr 2015
Deliverable Type:	RTD
Due Date of Delivery:	31 Oct 2014
Actual Date of Delivery:	15 Apr 2015
Nature of the Deliverable:	(R)eport
Dissemination Level:	Public

Abstract and Executive Summary:

This document specifies the mPlane architecture and protocol: in this executive summary, we attempt at providing a concise yet complete view, as a useful starting point for a walk-through of the mPlane architecture, additionally providing hyper-links to the specific sections of this document treating a selection of the most relevant items in the mPlane architecture.

This deliverable first provides a top-down introduction (Ch. 1.1) to all core mPlane concepts, providing insights about the main decisions to achieve flexibility and extensibility. This includes the definition of a schema-centric protocol, capable of iterative measurement and pledging for weak imperativeness to avoid deadlock situations. The chapter also introduces the different entities such as components, clients, probes, repositories supervisors, and reasoners. It finally describes relationships among entities, illustrating e.g., the interfaces to these entities, as well as the message types and typical exchange sequences.

The deliverable then delves into the details, building the mPlane protcol from bottom-up across three layers. The first layer consists of an information model for mPlane messages (Ch. 2). The information model includes an element registry (defining, e.g., name structure as well as several primitive types) that are expressed in several message of different types, such as capability specification result and event notification (including receipt, redemption, indirection and exception). Details of the messages are thoroughly illustrated -- including both low-level details e.g., temporal scope, parameters and metadata, as well as high-level principles such as message uniqueness and idempotence

The next layer is then represented by the representation and session protocols (Ch. 3), where details of the serialization of the mPlane information model using a JSON representation are illustrated, and examples of textual representations of element values are given. Session protocol description include secure access control schemes over HTTPS, over WebSockets and TLS or over SSH; only the first of these is presently implemented.

The deliverable then illustrates several example workflows (Ch. 4), including client-initiated processes, capability discovery phases, component-initiated worflow, callback control and indirect export, also covering how errors are handles in mPlane.

The remaining chapters (Ch. 5--8) are dedicated to regaining a complete vision of the architecture, so to fit detailed knowledge acquired so far (Ch. 2--4). Chapters covering for instance the role of the supervisor, the status of interoperable implementations, and data protection aspects, providing finally the core registry for completeness.

Keywords: architecture, use case, scenario, measurement, platform

SEVENTH FRAMEWORK PROGRAMME

Disclaimer

The information, documentation and figures available in this deliverable are written by the mPlane Consortium partners under EC co-financing (project FP7-ICT-318627) and does not necessarily reflect the view of the European Commission.

The information in this document is provided ``as is'', and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.

Contents

1 mPlane Architecture

mPlane is built around an architecture in which **components** provide network measurement services and access to stored measurement data which they advertise via **capabilities** completely describing these services and data. A **client** makes use of these capabilities by sending **specifications** that respond to them back to the components. Components may then either return **results** directly to the clients or sent to some third party via **indirect export** using an external protocol. The capabilities, specifications, and results are carried over the mPlane **protocol**, defined in detail in this document. An mPlane measurement infrastructure is built up from these basic blocks.

Components can be roughly classified into **probes** which generate measurement data and **repositories** which store and analyze measurement data, though the difference betweem a probe and a repository in the architecture is merely a matter of the capabilities it provides. Components can be pulled together into an infrastructure by a **supervisor**, which presents a client interface to subordinate components and a component interface to superordinate clients, aggregating capabilities into higher-level measurements and distributing specifications to perform them.

A client which provides automation support for measurement iteration in troubleshooting and root cause analysis is called a **reasoner**.

This arrangement is shown in schematic form in figure 1.1.

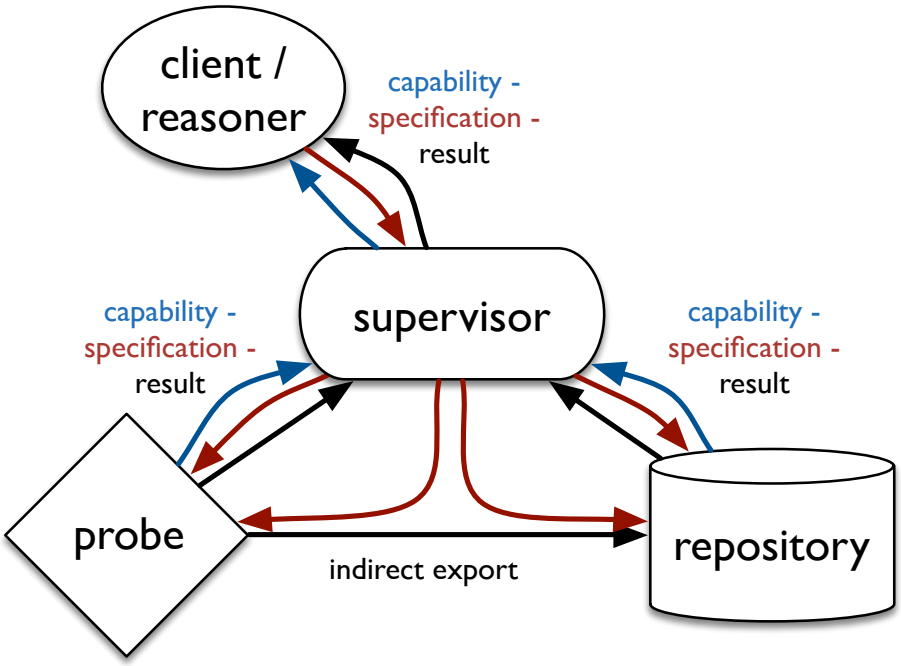

Figure 1.1: General arrangement of entities in the mPlane architecture

The mPlane protocol is, in essence, a self-describing, error- and delay-tolerant remote procedure call (RPC) protocol: each capability exposes an entry point in the API provided by the component; each specification embodies an API call; and each result returns the results of an API call.

1.1 Key architectural principles and features

mPlane differs from a simple RPC facility in several important ways, detailed in the subsections below. Each of these properties of the mPlane architecture and protocol follows from requirements which themselves were derived from an analysis of a set of specific use cases defined in mPlane Deliverable 1.1 [1], though the aim was to define an architecture applicable to a wider set of situations than these specific use cases.

1.1.1 Flexibility and extensibility

First, given the heterogeneity of the measurement tools and techniques applied, it is necessary for the protocol to be as *flexible* and *extensible* as possible. Therefore, the architecture in its simplest form consists of only two entities and one relationship, as shown in the figure 1.2: *n* clients connect to *m* components via the mPlane protocol. Anything which can speak the mPlane protocol and exposes capabilites thereby is a component; anything which can understand these capabilities and send specifications to invoke them is a client. Everything a component can do, from the point of view of mPlane, is entirely described by its capabilities. Capabilities are even used to expose optional internal features of the protocol itself, and provide a method for built-in protocol extensibility.

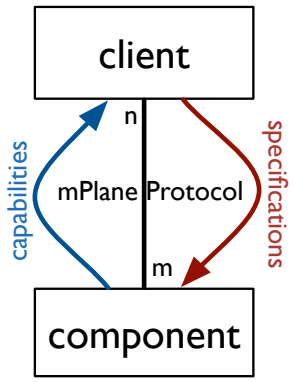

Figure 1.2: The simplest form of the mPlane architecture

1.1.2 Schema-centric measurement definition

Second, given the flexibility required above, the key to measurement interoperability is the comparison of data types. Each capability, specification, and result contains a *schema*, comprising the set of parameters required to execute a measurement or query and the columns in the data set that results. From the point of view of mPlane, the schema *completely* describes the measurement. This implies that when exposing a measurement using mPlane, the developer of a component must build each capability it advertises such that the semantics of the measurement are captured by the set of columns in its schema. The elements from which schemas can be built are captured in a type *registry*. The mPlane platform provides a core registry for common measurement use cases within the project, and the registry facility is itself fully extensible as well, for supporting new applications

without requiring central coordination beyond the domain or set of domains running the application.

1.1.3 Iterative measurement support

Third, the exchange of messages in the protocol was chosen to support *iterative* measurement in which the aggregated, high-level results of a measurement are used as input to a decision process to select the next measurement. Specifically, the protocol blends control messages (capabilities and specifications) and data messages (results) into a single workflow; this is shown in figure 1.3.

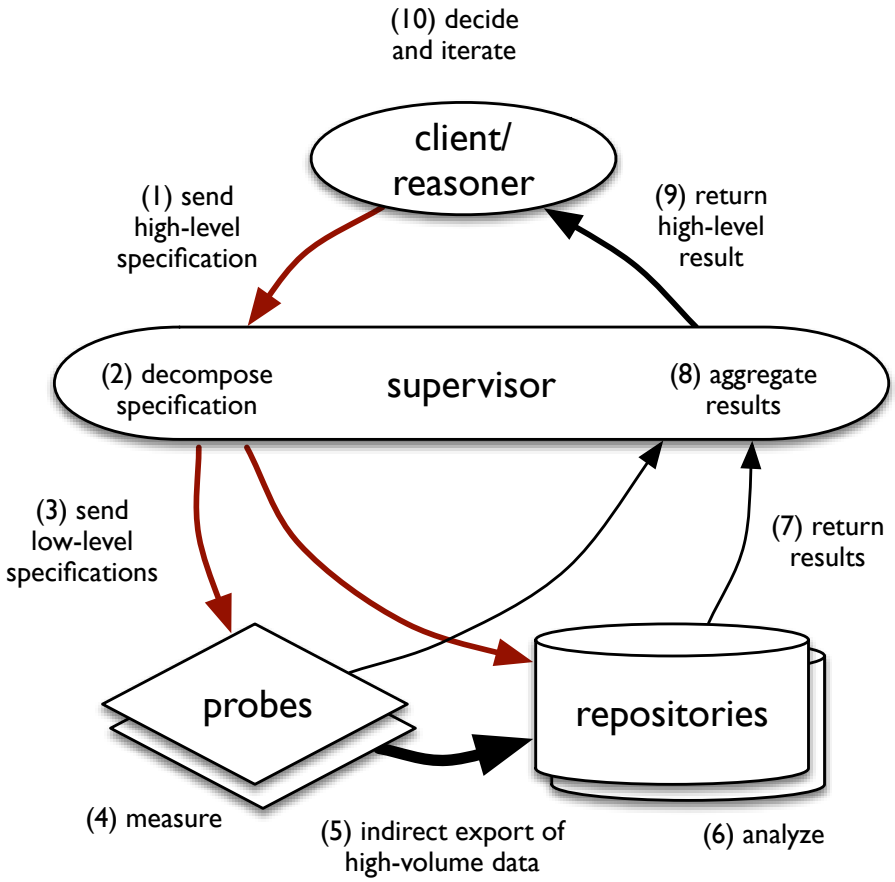

Figure 1.3: Iterative measurement in mPlane

1.1.4 Weak imperativeness

Fourth, the mPlane protocol is *weakly imperative*. A capability represents a willingness and an ability to perform a given measurement or execute a query, but not a guarantee or a reservation to do so. Likewise, a specification contains a set of parameters and a temporal scope for a measurement

a client wishes a component to perform on its behalf, but execution of specifications is best-effort. A specification is not an instruction which must result either in data or in an error. This property arises from our requirement to support large-scale measurement infrastructures with thousands of similar components, including resource- and connectivity-limited probes such as smartphones and customer-premises equipment (CPE) like home routers. These may be connected to a supervisor only intermittently. In this environment, the operability and conditions in which the probes find themselves may change more rapidly than can be practicably synchronized with a central supervisor; requiring reliable operation would compromise scalability of the architecture.

To support weak imperativeness, each message in the mPlane protocol is self-contained, and contains all the information required to understand the message. For instance, a specification contains the complete information from the capability which it responds to, and a result contains its specification. In essence, this distributes the state of the measurements running in an infrastructure across all components, and any state resynchronization that is necessary after a disconnect happens implicitly as part of message exchange. The failure of a component during a large-scale measurement can be detected and corrected after the fact, by examining the totality of the generated data.

This distribution of state throughout the measurement infrastructure carries with it a distribution of responsibility: a component holding a specification is responsible for ensuring that the measurement or query that specification describes is carried out, because the client or supervisor which has sent the specification does not necessarily keep any state for it.

Error handling in a weakly imperative environment is different to that in traditional RPC protocols. The exception facility provided by mPlane is designed only to report on failures of the handling of the protocol itself. Each component and client makes its best effort to interpret and process any authorized, well-formed mPlane protocol message it receives, ignoring those messages which are spurious or no longer relevant. This is in contrast with traditional RPC protocols, where even common exceptional conditions are signaled, and information about missing or otherwise defective data must be corrleated from logs about measurement control. This traditional design pattern is not applicable in infrastructures where the supervisor has no control over the functionality and availablility of its associated probes.

1.2 Entities and Relationships

The entities in the mPlane protocol and the relationships among them are described in more detail in the subsections below.

1.2.1 Components and Clients

Specifically, a **component** is any entity which implements the mPlane protocol specified within this document, advertises its capabilities and accepts specifications which request the use of those capabilities. The measurements, analyses, storage facilities and other services provided by a component are completely defined by its capabilities.

Conversely, a **client** is any entity which implements the mPlane protocol, receives capabilities published by one or more components, and sends specifications to those component(s) to perform

measurements and analysis.

Every interaction in the mPlane protocol takes place between a component and a client. Indeed, the simplest instantiation of the mPlane architecture consists of one or more clients taking capabilities from one or more components, and sending specifications to invoke those capabilities, as shown in figure 1.2. An mPlane domain may consist of as little as a single client and a single component. In this arrangement, mPlane provides a measurement-oriented RPC mechanism.

1.2.2 Probes and Repositories

Measurement components can be roughly divided into two categories: **probes** and **repositories**. Probes perform measurements, and repositories provide access to stored measurements, analysis of stored measurements, or other access to related external data sources. External databases and data sources (e.g., routing looking glasses, WHOIS services, DNS, etc.) can be made available to mPlane clients through repositories acting as gateways to these external sources, as well.

Note that this categorization is very rough: what a component can do is completely described by its capabilities, and some components may combine properties of both probes and repositories.

1.2.3 Supervisors and Federation

An entity which implements both the client and component interfaces can be used to build and federate domains of mPlane components. This **supervisor** is responsible for collecting capabilities from a set of components, and providing capabilities based on these to its clients. Application-specific algorithms at the supervisor aggregate the lower-level capabilities provided by these components into higher-level capabilities exposed to its clients. This arrangement is shown in figure 1.4.

The set of components which respond to specifications from a single supervisor is referred to as an mPlane **domain**. Domain membership is also determined by the issuer of the certificates identifying the clients, components, and supervisor, as detailed in section 3.2.2. Within a given domain, each client and component connects to only one supervisor. Underlying measurement components and clients may indeed participate in multiple domains, but these are separate entities from the point of view of the architecture. Interdomain measurement is supported by federation among supervisors: a local supervisor delegates measurements in a remote domain to that domain's supervisor, as shown in figure 1.5.

In addition to capability composition and specification decomposition, supervisors are responsible for client and component registration and authentication, as well as access control based on identity information provided by the session protocol (HTTPS, WebSockets, or SSH) in the general case.

Since the logic for aggregating control and data for a given application is very specific to that application, note that there is no *generic* supervisor implementation provided with the mPlane SDK.

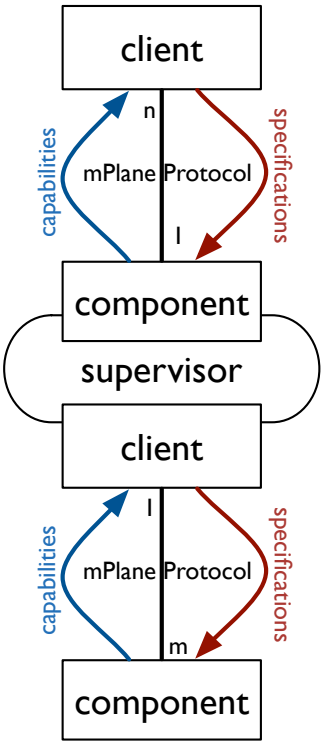

Figure 1.4: Simple mPlane architecture with a supervisor

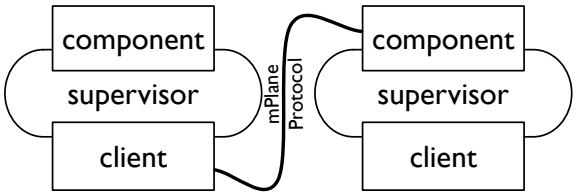

Figure 1.5: Federation between supervisors

1.2.4 Reasoner

Within an mPlane domain, a special client known as a **reasoner** may control automated or semi-automated iteration of measurements, e.g. working with a supervisor to iteratively run measurements using a set of components to perform root cause analysis. While the reasoner is key to the mPlane project, it is architecturally merely another client, though it will often be colocated with a supervisor for implementation convenience.

1.2.5 External interfaces to mPlane entities

The mPlane protocol specified in this document is designed for the exchange of control messages in an iterative measurement process, and the retrieval of low volumes of highly aggregated data, primarily that leads to decisions about subsequent measurements and/or a final determination.

For measurements generating large amounts of data (e.g. passive observations of high-rate links, or high-frequency active measurements), mPlane supports **indirect export**. For indirect export, a client or supervisor directs one component (generally a probe) to send results to another component (generally a repository). This indirect export protocol is completely external to the mPlane protocol; the client must only know that the two components support the same protocol and that the schema of the data produced by the probe matches that accepted by the repository. The typical example consists of passive mPlane-controlled probes exporting volumes of data (e.g., anonymized traces, logs, statistics), to an mPlane-accessible repository out-of-band. The use of out-of-band indirect export is justified to avoid serialization overhead, and to ensure fidelity and reliability of the transfer.

For exploratory analysis of large amounts of data at a repository, it is presumed that clients will have additional backchannel **direct access** beyond those interactions mediated by mPlane. For instance, a repository backed by a relational database could have a web-based graphical user interface that interacts directly with the database.

1.3 Message types and message exchange sequences

The basic messages in the mPlane protocol are capabilities, specifications, and results, as already described. The full protocol contains other message types as well. **Withdrawals** cancel capabilities (i.e., indicate that the component is no longer capable or willing to perform a given measurement) and **interrupts** cancel specifications (i.e., indicate that the component should stop performing the measurement). **Receipts** can be given in lieu of results for not-yet completed measurements or queries, and **redemptions** are used to retrieve results referred to by a receipt. If a component wants to delegate the handling of a specification to a different component, it sends an **indirection** to the client directing it to the target component. **Exceptions** (not shown in figure 1.6) can be sent by clients or components at any time to signal protocol-level errors to their peers.

In the nominal sequence, a capability leads to a specification leads to a result, where results may be transmitted by some other protocol. All the paths through the sequence of messages are shown infigure 1.6; message types are described in detail in section 2.2. In the diagram, solid lines mean a message is sent in reply to the previous message in sequence (i.e. a component sends a capability, and a client replies or follows with a specification), and dashed lines mean a message is sent as a followup (i.e., a component sends a capability, then sends a withdrawal to cancel that capability). Messages at the top of the diagram are sent by components, at the bottom by clients.

Separate from the sequence of messages, the mPlane protocol supports two connection establishment patterns:

- **Client-initiated** in which clients connect directly to components at known, stable, routable URLs. Client-initiated workflows are intended for use between clients and supervisors, for access to repositories, and for access to probes embedded within a network infrastructure.

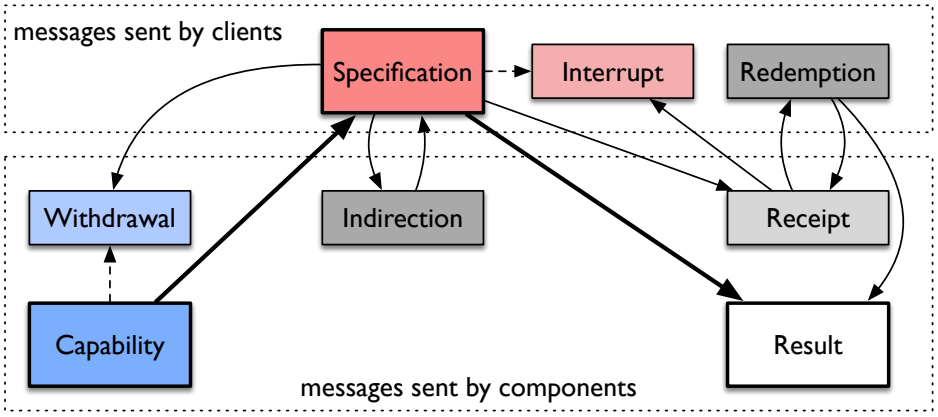

Figure 1.6: Potential sequences of messages in the mPlane protocol

- **Component-initiated** in which components initiate connections to clients. Component-initiated workflows are intended for use between components without stable routable addresses and supervisors, e.g. for small probes on embedded devices, mobile devices, or software probes embedded in browsers on personal computers behind network-address translators (NATs) or firewalls which prevent a client from establishing a connection to them.

Within a given mPlane domain, these patterns can be combined (along with indirect export and direct access) to facilitate complex interactions among clients and components according to the requirements imposed by the application and the deployment of components in the network.

1.4 A Cooperative measurement in an example mPlane domain

To illustrate how mPlane works within an example domain, consider figure 1.7. Here we see a client/reasoner, a supervisor, two probes, and a repository. The probes can perform simple latency and bandwidth measurements to a selected target, and send their results to the repository and storage for analysis. The repository can compare present with past measurements, determine whether a given target has higher latency or lower bandwidth than baseline.

To bootstrap the system, the probes and repository first publish their capabilities to the supervisor as shown in figure 1.7; here, each component knows the supervisor's address and establishes a connection to initiate capability advertisement.

Each probe sends a capability (`Ce') advertising the ability to measure bandwidth and latency to a target given that target, and to export than information to a repository via an external protocol. The repository sends a capability (`Cc') that it can collect data matching what the probes can export, as well as a capability (`C') advertising comparison to baseline. The supervisor registers these, then composes higher-level capabilities based upon them.

When a client or reasoner initiates a connection to the supervisor, these composed capabilities are advertised to it, as shown in figure 1.8. Here, the two higher-level capabilities offered by the supervisor are ``determine if a given target is nominal compared to baseline'' and ``show the recent measurements that deviate the most from the baseline''.

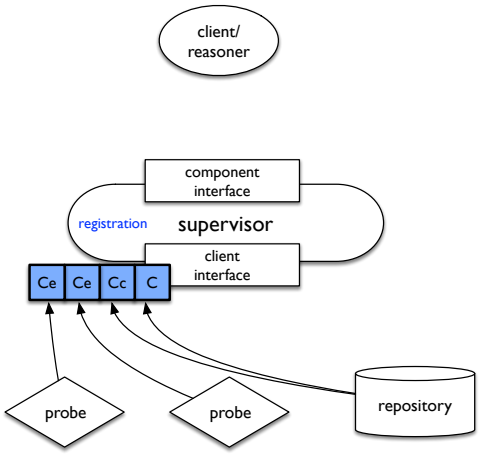

Figure 1.7: Capability advertisement and registration with a supervisor

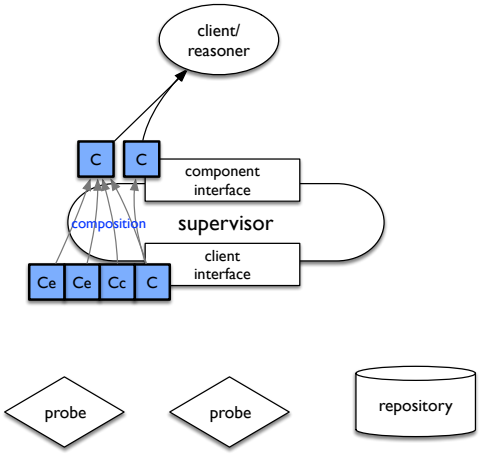

Figure 1.8: Capability composition at a supervisor

Suppose a user at the client decides to determine whether latency and bandwidth from the probes to a given target are within expected values. It sends a specification corresponding to the first capability made available by the supervisor to the supervisor, which then decomposes it into specifications to the probes and repository. First, it instructs the probes to take measurements and send them to the repository via indirect export, as shown in figure 1.9. Data export is shown as `Ex' in this diagram, noting that it uses some external protocol other than the mPlane protocol.

After enough measurements are completed, the supervisor then queries the repository to check that the measurements performed are within expected ranges given the history of measurements for that target, as shown in figure 1.10. Of course, the data from the probes becomes part of the repository's history for future queries about the target.

Note that not all interaction among components in an mPlane infrastructure must be mediated by the supervisor. This is particularly true of large-scale repositories, where (e.g.) visualization of large amounts of data may be done by accessing the repository's data directly using an external

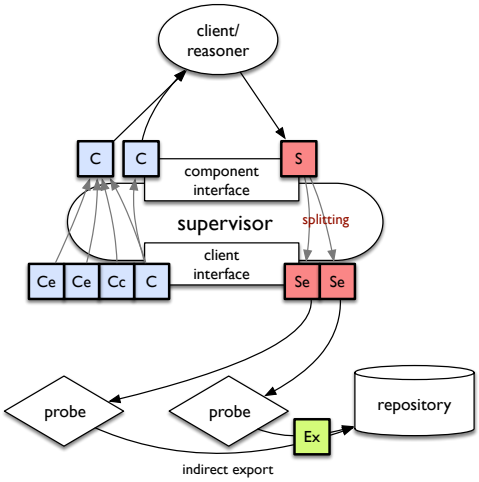

Figure 1.9: Specification delegation to probes by a supervisor

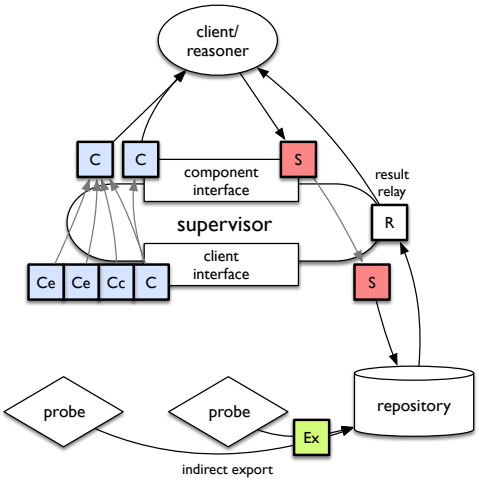

Figure 1.10: Querying for analysis of stored results at a repository

protocol, or by having the repository produce visualizations directly and providing these via HTTP.

1.5 Integrating measurement tools into mPlane

mPlane's flexibility and the self-description of measurements provided by the capability-specification-result cycle was designed to allow a wide variety of existing measurement tools, both probes and repositories, to be integrated into an mPlane domain. In both cases, the key to integration is to define a capability for each of the measurements the tool can perform or the queries the repository needs to make available within an mPlane domain. Each capability has a set of parameters -- information required to run the measurement or the query -- and a set of result columns -- information which the measurement or query returns. The parameters and result columns make up the mea-

surement's schema, and are chosen from an extensible registry of elements. Practical details are given in section 2.5.

1.6 From architecture to protocol specification

The remainder of this document builds the protocol specification based on this architecture from the bottom up. First, we define the protocol's information model from the element registry through the types of mPlane messages and the sections they are composed of. We then define a concrete representation of this information model using Javascript Object Notation (JSON, RFC 7159[2]), and define bindings to HTTP over TLS as a session protocol. Finally, we show how to construct workflows using the protocol to build up complex measurement infrastructures, and detail the responsibilities of an mPlane supervisor.

2 Protocol Information Model

The mPlane protocol is message-oriented, built on the representation- and session-protocol-independent exchange of messages between clients and components. This section describes the information model, starting from the element registry which defines the elements from which capabilities can be built, then detailing each type of message, and the sections that make these messages up. It then provides advice on using the information model to model measurements and queries.

2.1 Element Registry

An element registry makes up the vocabulary by which mPlane components and clients can express the meaning of parameters, metadata, and result columns for mPlane statements. A registry is represented as a JSON [2] object with the following keys:

- **registry-format**: currently `mplane-0`, determines the supported features of the registry format.
- **registry-uri**: the URI identifying the registry. The URI must be dereferenceable to retrieve the canonical version of this registry.
- **registry-revision**: a serial number starting with 0 and incremented with each revision to the content of the registry.
- **includes**: a list of URLs to retrieve additional registries from. Included registries will be evaluated in depth-first order, and elements with identical names will be replaced by registries parsed later.
- **elements**: a list of objects, each of which has the following three keys:
  - **name**: The name of the element.
  - **prim**: The name of the primitive type of the element, from the list of primitives in section 2.1.2.
  - **desc**: An English-language description of the meaning of the element.

Since the element names will be used as keys in mPlane messages, mPlane binds to JSON, and JSON mandates lowercase key names, element names must use only lowercase letters.

An example registry with two elements and no includes follows:

```
{ "registry-format": "mplane-0",
  "registry-uri", "http://ict-mplane.eu/registry/core",
  "registry-revision": 0,
  "includes": [],
  "elements": [
      { "name": "full.structured.name",
        "prim": "string",
        "desc": "A representation of foo..."
      },
      { "name": "another.structured.name",
        "prim": "string",
```

```
        "desc": "A representation of bar..."
      },
  ]
}
```

Fully qualified element names consist of the element's name as an anchor after the URI from which the element came, e.g. `http://ict-mplane.eu/registry/core#full.structured.name`. Elements within the type registry are considered globally equal based on their fully qualified names. However, within a given mPlane message, elements are considered equal based on unqualified names.

2.1.1 Structured Element Names

To ease understanding of mPlane type registries, element names are *structured* by convention; that is, an element name is made up of the following structural parts in order, separated by the dot (`.') character:

- **basename**: exactly one, the name of the property the element specifies or measures. All elements with the same basename describe the same basic property. For example, all elements with basename `source' relate to the source of a packet, flow, active measurement, etc.; and elements with basename `delay'' relate to the measured delay of an operation.
- **modifier**: zero or more, additional information differentiating elements with the same basename from each other. Modifiers may associate the element with a protocol layer, or a particular variety of the property named in the basename. All elements with the same basename and modifiers refer to exactly the same property. Examples for the `delay` basename include `oneway` and `twoway`, differentiating whether a delay refers to the path from the source to the destination or from the source to the source via the destination; and `icmp` and `tcp`, describing the protocol used to measure the delay.
- **units**: zero or one, present if the quantity can be measured in different units.
- **aggregation**: zero or one, if the property is a metric derived from multiple singleton measurements. Supported aggregations are:
- `min`: minimum value
- `max`: maximum value
- `mean`: mean value
- `sum`: sum of values
- `NNpct` (where NN is a two-digit number 01-99): percentile
- `median`: shorthand for and equivalent to `50pct`.
- `count`: count of values aggregated

When mapping mPlane structured names into contexts in which dots have special meaning (e.g. SQL column names or variable names in many programming languages), the dots may be replaced by underscores ('_'). When using external type registries (e.g. the IPFIX Information Element Registry), element names are not necessarily structured.

2.1.2 Primitive Types

The mPlane protocol supports the following primitive types for elements in the type registry:

- **string**: a sequence of Unicode characters
- **natural**: an unsigned integer
- **real**: a real (floating-point) number
- **bool**: a true or false (boolean) value
- **time**: a timestamp, expressed in terms of UTC. The precision of the timestamp is taken to be unambiguous based on its representation.
- **address**: an identifier of a network-level entity, including an address family. The address family is presumed to be implicit in the format of the message, or explicitly stored. Addresses may represent specific endpoints or entire networks.
- **url**: a uniform resource locator

2.1.3 Augmented Registry Information

Additional keys beyond **prim**, **desc**, and **name** may appear in an mPlane registry to augment information about each element; these are not presently used by the SDK's information model but may be used by software built around the SDK.

Elements in the core registry at `http://ict-mplane.eu/registry/core` may contain the following augmented registry keys:

- **units**: If applicable, units in which the element is expressed; equal to the units part of a structured name if present.
- **ipfix-eid**: The element ID of the equivalent IPFIX (RFC 7011 [3]) Information Element.
- **ipfix-pen**: The SMI Private Enterprise Number of the equivalent IPFIX Information Element, if any.

2.2 Message Types

Workflows in mPlane are built around the *capability - specification - result* cycle. Capabilities, specifications, and results are kinds of **statements**: a capability is a statement that a component can perform some action (generally a measurement); a specification is a statement that a client would like a component to perform the action advertised in a capability; and a result is a statement that a component measured a given set of values at a given point in time according to a specification.

Other types of messages outside this nominal cycle are referred to as **notifications**. Types of notifications include Withdrawals, Interrupts, Receipts, Redemptions, Indirections, and Exceptions. These notify clients or components of conditions within the measurement infrastructure itself, as opposed to directly containing information about measurements or observations.

Messages may also be grouped together into a single **envelope** message. Envelopes allow multiple messages to be represented within a single message, for example multiple Results pertaining to the

same Receipt; and multiple Capabilities or Specifications to be transferred in a single transaction in the underlying session protocol.

The following types of messages are supported by the mPlane protocol:

2.2.1 Capability and Withdrawal

A **capability** is a statement of a component's ability and willingness to perform a specific operation, conveyed from a component to a client. It does not represent a guarantee that the specific operation can or will be performed at a specific point in time.

A **withdrawal** is a notification of a component's inability or unwillingness to perform a specific operation. It cancels a previously advertised capability. A withdrawal can also be sent in reply to a specification which attempts to invoke a capability no longer offered.

2.2.2 Specification and Interrupt

A **specification** is a statement that a component should perform a specific operation, conveyed from a client to a component. It can be conceptually viewed as a capability whose parameters have been filled in with values.

An **interrupt** is a notification that a component should stop performing a specific operation, conveyed from client to component. It terminates a previously sent specification. If the specification uses indirect export, the indirect export will simply stop running. If the specification has pending results, those results are returned in response to the interrupt.

2.2.3 Result

A **result** is a statement produced by a component that a particular measurement was taken and the given values were observed, or that a particular operation or analysis was performed and a the given values were produced. It can be conceptually viewed as a specification whose result columns have been filled in with values. Note that, in keeping with the stateless nature of the mPlane protocol, a result contains the full set of parameters from which it was derived.

Note that not every specification will lead to a result being returned; for example, in case of indirect export, only a receipt which can be used for future interruption will be returned, as the results will be conveyed to a third component using an external protocol.

2.2.4 Receipt and Redemption

A **receipt** is returned instead of a result by a component in response to a specification which either:

- will never return results, as it initiated an indirect export, or
- will not return results immediately, as the operation producing the results will have a long run time.

Receipts have the same content specification they are returned for. A component may optionally add a **token** section, which can be used in future redemptions or interruptions by the client. The content of the token is an opaque string generated by the component.

A **redemption** is sent from a client to a component for a previously received receipt to attempt to retrieve delayed results. It may contain only the **token** section, or all sections of the received receipt.

2.2.5 Indirection

An **indirection** is returned instead of a result by a component to indicate that the client should contact another component for the desired result.

2.2.6 Exception

An **exception** is sent from a client to a component or from a component to a client to signal an exceptional condition within the infrastructure itself. They are not meant to signal exceptional conditions within a measurement performed by a component; see section 4.4 for more. An exception contains only two sections: an optional **token** referring back to the message to which the exception is related (if any), and a **message** section containing free-form, preferably human readable information about the exception.

2.2.7 Envelope

An **envelope** is used to contain other messages. Message containment is necessary in contexts in which multiple mPlane messages must be grouped into a single transaction in the underlying session protocol. It is legal to group any kind of message, and to mix messages of different types, in an envelope. However, in the current revision of the protocol, envelopes are primarily intended to be used for three distinct purposes:

- To return multiple results for a single receipt or specification if appropriate (e.g., if a specification has run repeated instances of a measurement on a schedule).
- To group multiple capabilities together within a single message (e.g., all the capabilities a given component has).
- To group multiple specifications into a single message (e.g., to simultaneously send a measurement specification along with a callback control specification).

2.3 Message Sections

Each message is made up of sections, as described in the subsection below. The following table shows the presence of each of these sections in each of the message types supported by mPlane: ``req.'' means the section is required, ``opt.'' means it is optional; see the subsection on each message section for details.

Section	Capability	Specification	Result	Receipt	Envelope
Verb	req.	req.	req.	req.	
Content Type					req.
`version`	req.	req.	req.	req.	req.
`registry`	req.	req.	req.	opt.	
`label`	opt.	opt.	opt.	opt.	opt.
`when`	req.	req.	req.	req.	
`parameters`	req./token	req.	req.	opt./token	
`metadata`	opt./token	opt.	opt.	opt./token	
`results`	req./token	req.	req.	opt./token	
`resultvalues`			req.		
`export`	opt.	opt.	opt.	opt.	
`link`	opt.	opt.			
`token`	opt.	opt.	opt.	opt.	opt.
`contents`					req.

Withdrawals and indirections take the same sections as capabilities; and redemptions and interrupts take the same sections as receipts. Exceptions are not shown in this table.

2.3.1 Message Type and Verb

The **verb** is the action to be performed by the component. The following verbs are supported by the base mPlane protocol, but arbitrary verbs may be specified by applications:

- `measure`: Perform a measurement
- `query`: Query a database about a past measurement
- `collect`: Receive results via indirect export
- `callback`: Used for callback control in component-initiated workflows

In the JSON representation of mPlane messages, the verb is the value of the key corresponding to the message's type, represented as a lowercase string (e.g. `capability`, `specification`, `result` and so on).

Roughly speaking, probes implement `measure` capabilities, and repositories implement `query` and `collect` capabilities. Of course, any single component can implement capabilities with any number of different verbs.

Within the SDK, the primary difference between `measure` and `query` is that the temporal scope of a `measure` specification is taken to refer to when the measurement should be scheduled, while the temporal scope of a `query` specification is taken to refer to the time window (in the past) of a query.

Envelopes have no verb; instead, the value of the `envelope` key is the kind of messages the envelope contains, or `message` if the envelope contains a mixture of different unspecified kinds of messages.

2.3.2 Version

The `version` section contains the version of the mPlane protocol to which the message conforms, as an integer serially incremented with each new protocol revision. This section is required in all messages. This document describes version 1 of the protocol.

2.3.3 Registry

The `registry` section contains the URL identifying the element registry used by this message, and from which the registry can be retrieved. This section is required in all messages containing element names (statements, and receipts/redemptions/interrupts not using tokens for identification; see the `token` section). The default core registry for mPlane is identified by:

`http://ict-mplane.eu/registry/core`.

2.3.4 Label

The `label` section of a statement contains a human-readable label identifying it, intended solely for use when displaying information about messages in user interfaces. Results, receipts, redemptions, and interrupts inherit their label from the specification from which they follow; otherwise, client and component software can arbitrarily assign labels . The use of labels is optional in all messages, but as labels do greatly ease human-readability of arbitrary messages within user interfaces, their use is recommended.

mPlane clients and components should **never** use the label as a unique identifier for a message, or assume any semantic meaning in the label -- the test of message equality and relatedness is always based upon the schema and values as in section 2.4.

2.3.5 Temporal Scope (When)

The `when` section of a statement contains its **temporal scope**.

A temporal scope refers to when a measurement can be run (in a capability), when it should be run (in a specification), or when it was run (in a result). Temporal scopes can be either absolute or relative, and may have an optional period, referring to how often single measurements should be taken.

The general form of a temporal scope (in BNF-like syntax) is as follows:

```
simple-when = <singleton> |            # A single point in time
              <range> |                # A range between two points in time
              <range> ' / ' <duration> # A range with a period

singleton = <iso8601> | # absolute singleton
            'now'       # relative singleton
```

```
range = <iso8601> ' ... ' <iso8601> | # absolute range
        <iso8601> ' + ' <duration> |  # relative range
        'now' ' ... ' <iso8061> |     # definite future
        'now' ' + ' <duration> |      # relative future
        <iso8601> ' ... ' 'now' |     # definite past
        'past ... now' |              # indefinite past
        'now ... future' |            # indefinite future
        <iso8601> ' ... ' 'future' |  # absolute indefinite future
        'past ... future' |           # forever

duration = [ <n> 'd' ] # days
           [ <n> 'h' ] # hours
           [ <n> 'm' ] # minute
           [ <n> 's' ] # seconds

iso8601 = <n> '-' <n> '-' <n> [' ' <n> ':' <n> ':' <n> [ '.' <n> ]
```

All absolute times are **always** given in UTC and expressed in ISO8601 format with variable precision.

In capabilities, if a period is given it represents the *minimum* period supported by the measurement; this is done to allow rate limiting. If no period is given, the measurement is not periodic. A capability with a period can only be fulfilled by a specification with period greater than or equal to the period in the capability. Conversely, a capability without a period can only be fulfilled by a specification without a period.

Within a result, only absolute ranges are allowed within the temporal scope, and refers to the time range of the measurements contributing to the result. Note that the use of absolute times here implies that the components and clients within a domain should have relatively well-synchronized clocks, e.g., to be synchronized using the Network Time Protocol (RFC 5905 [10]) in order for results to be temporally meaningful.

So, for example, an absolute range in time might be expressed as:

```
when: 2009-02-20 13:02:15 ... 2014-04-04 04:27:19
```

A relative range covering three and a half days might be:

```
when: 2009-04-04 04:00:00 + 3d12h
```

In a specification for running an immediate measurement for three hours every seven and a half minutes:

```
when: now + 3h / 7m30s
```

In a capability noting that a repository can answer questions about the past:

```
when: past ... now.
```

In a specification requesting that a measurement run from a specified point in time until interrupted:

```
when: 2017-11-23 18:30:00 ... future
```

2.3.5.1 Repeating Measurements

Within specifications, the temporal scope can be extended to support **repeated measurement**. A repeated specification is conceptually equivalent to a specification that is sent from the client to the component once, then retained at the component and initiated multiple times.

The general form of a temporal scope in a repeated specification is as follows (BNF-like syntax):

```
repeated-when = # implicit inner scope of now
                'repeat' <outer-when> |
                # simple range/period
                'repeat' <outer-when> '{' <inner-when> '}' |
                # with crontab
                'repeat' <range> 'cron' <crontab> '{' <inner-when> '}'

outer-when = <range> ' / ' <duration>

inner-when = 'now' |
             'now' ' + ' <duration> |
             'now' ' + ' <duration> / <duration>

crontab = <seconds> <minutes> <hours> <days-of-month> <days-of-week> <months>

seconds = '*' | <seconds-or-minutes-list>
minutes = '*' | <seconds-or-minutes-list>
seconds-or-minutes-list = <n> [ ',' <seconds-or-minutes-list> ] # 0<=n<60

hours = '*' | <hours-list>
hours-list = <n> [ ',' <hour-list> ] # 0<=n<24

days-of-month = '*' | <days-of-month-list>
days-of-month-list = <n> [ ',' <days-of-month-list> ] # 0<n<=31

days-of-week = '*' | <days-of-week-list>
days-of-week-list = <n> [ ',' <days-of-week-list> ] # 0<=n<=7
                    # 0 = Sunday, 1 = Monday, ..., 7 = Sunday

months = '*' | <months-list>
months-list = <n> [ ',' <months-list> ] # 0<n<=12

when = <simple-when> | <repeated-when>
```

A repeated specification consists of an *outer* temporal specification that governs how often and for how long the specification will repeat, and an *inner* temporal specification which applies to each individual repetition. The inner temporal specification must *always* be relative to the current time, i.e. the time of initiated of the repeated specification. If the inner temporal specification is omitted, the specification is presumed to have the relative singleton temporal scope of `now`.

A repeated specification can have a cron-like schedule. In this case the *outer* temporal specification only consists of a *range* scope to determine the time frame in which the cron-like schedule is valid. The *crontab* states the seconds, minutes, hours, days of the week, days of the month, and months at which the specification will repeat. An asterisk means to repeat at all legal values for that field. The specification is only repeated if all fields match.

Submitting a repeated specification will still result in a single receipt, or in multiple results. These multiple results, resulting either directly from a single repeated specification, or from the a redemption of a receipt resulting from a repeated specification, are grouped in an envelope message.

For example, a repeated specification to take measurements every second for five minutes, repeating once an hour indefinitely would be:

```
when: repeat now ... future / 1h { now + 5m / 1s }
```

This repeated specification is equivalent to the repeated submission of the same specification with a temporal scope of `when: { now + 5m / 1s }` once an hour until the specification is cancelled with an interrupt notification.

As a second example, a repeated specification to take measurements every second for five minutes, repeating every half hour within a specific timeframe would be:

```
when: repeat 2014-01-01 13:00:00 ... 2014-06-01 14:00:00 / 30m { now + 5m / 1s }
```

Likewise, this repeated specification is equivalent to the submission of the same specification with a temporal scope of `when: { now + 5m / 1s }` at `2014-01-01 13:00:00`, `2014-01-01 13:30:00`, `2014-01-01 14:00:00`, `2014-01-01 14:30:00`, and so on =, until (and including) `2014-06-01 13:30:00` and `2014-06-01 14:00:00`.

A repeated specification taking singleton measurements every hour indefinitely with an implicit inner temporal specification:

```
when: repeat now ... future / 1h
```

equivalent to submitting a specification with the temporal scope `now` hourly forever until interrupted.

A crontab specification which is repeated on the first Monday of each month measuring every hour on that day for 5 minutes would be: `when: repeat now ... future cron 0 0 * 1,2,3,4,5,6,7 1 * { now + 5m }`

A repeated specification to take measurements each day of the year at midnight would be: `when: repeat now ... future cron 0 0 0 * * *`

2.3.6 Parameters

The `parameters` section of a message contains an ordered list of the **parameters** for a given measurement: values which must be provided by a client to a component in a specification to convey the specifics of the measurement to perform. Each parameter in an mPlane message is a **key-value pair**, where the key is the name of an element from the element registry. In specifications and results, the value is the value of the parameter. In capabilities, the value is a **constraint** on the possible values the component will accept for the parameter in a subsequent specification.

Four kinds of constraints are currently supported for mPlane parameters:

- No constraint: all values are allowed. This is signified by the special constraint string `*'.
- Single value constraint: only a single value is allowed. This is intended for use for capabilities which are conceivably configurable, but for which a given component only supports a single value for a given parameter due to its own out-of-band configuration or the permissions of the client for which the capability is valid. For example, the source address of an active measurement of a single-homed probe might be given as `source.ip4: 192.0.2.19'.
- Set constraint: multiple values are allowed, and are explicitly listed, separated by the `,' character. For example, a multi-homed probe allowing two potential source addresses on two different networks might be given as `source.ip4: 192.0.2.19, 192.0.3.21'.
- Range constraint: multiple values are allowed, between two ordered values, separated by the special string `...'. Range constraints are inclusive. A measurement allowing a restricted range of source ports might be expressed as `source.port: 32768 ... 65535'
- Prefix constraint: multiple values are allowed within a single network, as specified by a network address and a prefix. A prefix constraint may be satisfied by any network of host address completely contained within the prefix. An example allowing probing of any host within a given /24 might be `destination.ip4: 192.0.2.0/24'.

Parameter and constraint values must be a representation of an instance of the primitive type of the associated element.

2.3.7 Metadata

The `metadata` section contains measurement **metadata**: key-value pairs associated with a capability inherited by its specification and results. Metadata can also be thought of as immutable parameters. This is intended to represent information which can be used to make decisions at the client as to the applicability of a given capability (e.g. details of algorithms used or implementation-specific information) as well as to make adjustments at post-measurement analysis time when contained within results.

An example metadata element might be `measurement.identifier: qof', which identifies the underlying tool taking measurements, such that later analysis can correct for known peculiarities in the implementation of the tool. Another example might be `location.longitude = 8.55272', which while not particularly useful for analysis purposes, can be used to draw maps of measurements.

2.3.8 Result Columns and Values

Results are represented using two sections: `results` which identify the elements to be returned by the measurement, and `resultvalues` which contains the actual values. `results` appear in all statements, while `resultvalues` appear only in result messages.

The `results` section contains an ordered list of **result columns** for a given measurement: names of elements which will be returned by the measurement. The result columns are identified by the names of the elements from the element registry.

The `resultvalues` section contains an ordered list of ordered lists (or, rather, a two dimensional array) of values of results for a given measurement, in row-major order. The columns in the result

values appear in the same order as the columns in the `results` section.

Values for each column must be a representation of an instance of the primitive type of the associated result column element.

2.3.9 Export

The `export` section contains a URL or partial URL for **indirect export**. Its meaning depends on the kind and verb of the message:

- For capabilities with the `collect` verb, the `export` section contains the URL of the collector which can accept indirect export for the schema defined by the `parameters` and `results` sections of the capability, using the protocol identified by the URL's schema.
- For capabilities with any verb other than `collect`, the `export` section contains either the URL of a collector to which the component can indirectly export results, or a URL schema identifying a protocol over which the component can export to arbitrary collectors.
- For specifications with any verb other than `collect`, the `export` section contains the URL of a collector to which the component should indirectly export results. A receipt will be returned for such specifications.

If a component can indirectly export or indirectly collect using multiple protocols, each of those protocols must be identified by its own capability; capabilities with an `export` section can only be used by specifications with a matching `export` section.

The special export schema `mplane-https` implies that the exporter will POST mPlane result messages to the collector at the specified URL. All other export schemas are application-specific, and the mPlane protocol implementation is only responsible for ensuring the schemas and protocol identifiers match between collector and exporter.

2.3.10 Link

The `link` section contains the URL to which messages in the next step in the workflow (i.e. a specification for a capability, a result or receipt for a specification) can be sent, providing indirection. The link URL must currently have the schema `mplane-https`, and refers to posting of messages via HTTP `POST`.

If present in a capability, the client must `POST` specifications for the given capability to the component at the URL given in order to use the capability, as opposed to simply posting them to the known or assumed URL for a component. If present in a specification, the component must `POST` results for the given specification back to the client at the URL given. See the section on workflows below for details.

If present in an indirection message returned for a specification by a component, the client must send the specification to the component at the URL given in the link in order to retrieve results or initiate measurement.

2.3.11 Token

The `token` section contains an arbitrary string by which a message may be identified in subsequent communications in an abbreviated fashion. Unlike labels, tokens are not necessarily intended to be human-readable; instead, they provide a way to reduce redundancy on the wire by replacing the parameters, metadata, and results sections in messages within a workflow, at the expense of requiring more state at clients and components. Their use is optional.

Tokens are scoped to the association between the component and client in which they are first created; i.e., at a component, the token will be associated with the client's identity, and vice-versa at a client. Tokens should be created with sufficient entropy to avoid collision from independent processes at the same client or token reuse in the case of client or component state loss at restart.

If a capability contains a token, it may be subsequently withdrawn by the same component using a withdrawal containing the token instead of the parameters, metadata, and results sections.

If a specification contains a token, it may be answered by the component with a receipt containing the token instead of the parameters, metadata, and results sections. A specification containing a token may likewise be interrupted by the client with an interrupt containing the token. A component must not answer a specification with a token with a receipt or result containing a different token, but the token may be omitted in subsequent receipts and results.

If a receipt contains a token, it may be redeemed by the same client using a redemption containing the token instead of the parameters, metadata, and results sections.

When grouping multiple results from a repeating specification into an envelope, the envelope may contain the token of the repeating specification.

2.3.12 Contents

The `contents` section appears only in envelopes, and is an ordered list of messages. If the envelope's kind identifies a message kind, the contents may contain only messages of the specified kind, otherwise if the kind is `message`, the contents may contain a mix of any kind of message.

2.4 Message uniqueness and idempotence

Messages in the mPlane protocol are intended to support **state distribution**: capabilities, specifications, and results are meant to be complete declarations of the state of a given measurement. In order for this to hold, it must be possible for messages to be uniquely identifiable, such that duplicate messages can be recognized. With one important exception (i.e., specifications with relative temporal scopes), messages are *idempotent*: the receipt of a duplicate message at a client or component is a null operation.

2.4.1 Message schema

The combination of elements in the `parameters` and `results` sections, together with the registry from which these elements are drawn, is referred to as a message's **schema**. The schema of a mea-

surement can be loosely thought of as the definition of the table, rows of which the message represents.

The schema contributes not only to the identity of a message, but also to the semantic interpretation of the parameter and result values. The meanings of element values in mPlane are dependent on the other elements present in the message; in other words, the key to interpreting an mPlane message is that *the unit of semantic identity is a message*. For example, the element `destination.ip4' as a parameter means ``the target of a given active measurement'' when together with elements describing an active metric (e.g. `delay.twoway.icmp.us'), but ``the destination of the packets in a flow'' when together with other elements in result columns describing a passively-observed flow.

The interpretation of the semantics of an entire message is application-specific. The protocol does not forbid the transmission of messages representing semantically meaningless or ambiguous schemas.

2.4.2 Message identity

A message's identity is composed of its schema, together with its temporal scope, metadata, parameter values, and indirect export properties. Concretely, the full content of the `registry`, `when`, `parameters`, `metadata`, `results`, and `export` sections taken together comprise the message's identity.

One convenience feature complicates this somewhat: when the temporal scope is not absolute, multiple specifications may have the same literal temporal scope but refer to different measurements. In this case, the current time at the client or component when a message is invoked must be taken as part of the message's identity as well. Implementations may use hashes over the values of the message's identity sections to uniquely identify messages; e.g. to generate message tokens.

2.5 Designing measurement and repository schemas

As noted, the key to integrating a measurement tool into an mPlane infrastructure is properly defining the schemas for the measurements and queries it performs, then defining those schemas in terms of mPlane capabilities. Specifications and results follow naturally from capabilities, and the mPlane SDK allows Python methods to be bound to capabilities in order to execute them. A schema should be defined such that the set of parameters, the set of result columns, and the verb together naturally and uniquely define the measurement or the query being performed. For simple metrics, this is achieved by encoding the entire meaning of the metric in its name. For example, `delay.twoway.icmp.us` as a result column together with `source.ip4` and `destination.ip4` as parameters uniquely defines a single ping measurement, measured via ICMP, expressed in microseconds.

Aggregate measurements are defined by returning metrics with aggregations, e.g. `delay.twoway.icmp.min.us`, `delay.twoway.icmp.max.us`, `delay.twoway.icmp.mean.us`, and `delay.twoway.icmp.count.us` as result columns represent aggregate ping measurements with multiple samples.

Note that mPlane results may contain multiple rows. In this case, the parameter values in the result, taken from the specification, apply to all rows. In this case, the rows are generally differentiated by the values in one or more result columns; for example, the `time` element can be used to represent time series, or the `hops.ip` different elements along a path between source and destination, as in

a traceroute measurement.

For measurements taken instantaneously, the verb `measure` should be used; for direct queries from repositories, the verb `query` should be used. Other actions that cannot be differentiated by schema alone should be differentiated by a custom verb.

When integrating a repository into an mPlane infrastructure, only a subset of the queries the repository can perform will generally be exposed via the mPlane interface. Consider a generic repository which provides an SQL interface for querying data; wrapping the entire set of possible queries in specific capabilities would be impossible, while providing direct access to the underlying SQL (for instance, by creating a custom registry with a `query.sql` string element to be used as a parameter) would make it impossible to differentiate capabilities by schema (thereby making the interoperability benefits of mPlane integration pointless). Instead, specific queries to be used by clients in concert with capabilities provided by other components are each wrapped within a separate capability, analogous to stored procedure programming in many database engines. Of course, clients which do speak the precise dialect of SQL necessary can integrate directly with the repository separate from the capabilities provided over mPlane.

3 Representations and Session Protocols

The mPlane protocol is built atop an abstract data model in order to support multiple representations and session protocols. The canonical representation supported by the present SDK involves JSON [2] objects transported via HTTP (RFC 7230 [6]) over TLS (RFC 5246 [4]) (commonly known as HTTPS).

3.1 JSON representation

In the JSON representation, an mPlane message is a JSON object, mapping sections by name to their contents. The name of the message type is a special section key, which maps to the message's verb, or to the message's content type in the case of an envelope.

Each section name key in the object has a value represented in JSON as follows:

- `version` : an integer identifying the mPlane protocol version used by the message.
- `registry` : a URL identifying the registry from which element names are taken.
- `label` : an arbitrary string.
- `when` : a string containing a temporal scope, as described in section 2.3.5.
- `parameters` : a JSON object mapping (non-qualified) element names, either to constraints or to parameter values, as appropriate, and as described in section 2.3.6.
- `metadata` : a JSON object mapping (non-qualified) element names to metadata values.
- `results` : an array of element names.
- `resultvalues` : an array of arrays of element values in row major order, where each row array contains values in the same order as the element names in the `results` section.
- `export` : a URL for indirect export.
- `link` : a URL for message indirection.
- `token` : an arbitrary string.
- `contents` : an array of objects containing messages.

3.1.1 Textual representations of element values

Each primitive type is represented as a value in JSON as follows, following the Textual Representation of IPFIX Abstract Data Types defined in RFC7373 [11].

Natural and real values are represented in JSON using native JSON representation for numbers.

Booleans are represented by the reserved words `true` and `false`.

Strings and URLs are represented as JSON strings subject to JSON escaping rules.

Addresses are represented as dotted quads for IPv4 addresses as they would be in URLs, and canonical IPv6 textual addresses as in section 2.2 of RFC 4291 [7] as updated by section 4 of RFC 5952 [8]. When representing networks, addresses may be suffixed as in CIDR notation, with a `/' character followed by the mask length in bits n, provided that the least significant $32-n$ or $128-n$ bits of the address are zero, for IPv4 and IPv6 respectively.

Timestamps are represented in RFC 3339 [9] and ISO 8601, with two important differences. First, all mPlane timestamps are expressed in terms of UTC, so time zone offsets are neither required nor supported, and are always taken to be 0. Second, fractional seconds are represented with a variable number of digits after an optional decimal point after the fraction.

3.1.2 Example mPlane capabilities and specifications

To illustrate how mPlane messages are encoded, we consider first two capabilities for a very simple application -- ping -- as mPlane JSON capabilities. The following capability states that the component can measure ICMP two-way delay from 192.0.2.19 to anywhere on the IPv4 Internet, with a minimum delay between individual pings of 1 second, returning aggregate statistics:

```
{
  "capability": "measure",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "ping-aggregate",
  "when":       "now ... future / 1s",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "*"},
  "results":    ["delay.twoway.icmp.us.min",
                 "delay.twoway.icmp.us.mean",
                 "delay.twoway.icmp.us.50pct",
                 "delay.twoway.icmp.us.max",
                 "delay.twoway.icmp.count"]
}
```

In contrast, the following capability would return timestamped singleton delay measurements given the same parameters:

```
{
  "capability": "measure",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "ping-singletons",
  "when":       "now ... future / 1s",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "*"},
  "results":    ["time",
                 "delay.twoway.icmp.us"]
}
```

A specification is merely a capability with filled-in parameters, e.g.:

```
{
```

```
  "specification":  "measure",
  "version":        0,
  "registry":       "http://ict-mplane.eu/registry/core",
  "label":          "ping-aggregate-three-thirtythree",
  "token":          "0f31c9033f8fce0c9be41d4942c276e4",
  "when":           "now + 30s / 1s",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "192.0.3.33"},
  "results":    ["delay.twoway.icmp.us.min",
                 "delay.twoway.icmp.us.mean",
                 "delay.twoway.icmp.us.50pct",
                 "delay.twoway.icmp.us.max",
                 "delay.twoway.icmp.count"]
}
```

Results are merely specifications with result values filled in and an absolute temporal scope:

```
{
  "result":         "measure",
  "version":        0,
  "registry":       "http://ict-mplane.eu/registry/core",
  "label":          "ping-aggregate-three-thirtythree",
  "token":          "0f31c9033f8fce0c9be41d4942c276e4",
  "when":           "2014-08-25 14:51:02.623 ... 2014-08-25 14:51:32.701 / 1s",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "192.0.3.33"},
  "results":    ["delay.twoway.icmp.us.min",
                 "delay.twoway.icmp.us.mean",
                 "delay.twoway.icmp.us.50pct",
                 "delay.twoway.icmp.us.max",
                 "delay.twoway.icmp.count"],
  "resultvalues": [ [ 23901,
                      29833,
                      27619,
                      66002,
                      30] ]
}
```

More complex measurements can be modeled by mapping them back to tables with multiple rows. For example, a traceroute capability would be defined as follows:

```
{
  "capability": "measure",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "traceroute",
```

```
  "when":       "now ... future / 1s",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "*",
                 "hops.ip.max": "0..32"},
  "results":    ["time",
                 "intermediate.ip4",
                 "hops.ip",
                 "delay.twoway.icmp.us"]
}
```

with a corresponding specification:

```
{
  "specification": "measure",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "traceroute-three-thirtythree",
  "token":      "2f4123588b276470b3641297ae85376a",
  "when":       "now",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "192.0.3.33",
                 "hops.ip.max": 32},
  "results":    ["time",
                 "intermediate.ip4",
                 "hops.ip",
                 "delay.twoway.icmp.us"]
}
```

and an example result:

```
{
  "result": "measure",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "traceroute-three-thirtythree",
  "token":      "2f4123588b276470b3641297ae85376a,
  "when":       "2014-08-25 14:53:11.019 ... 2014-08-25 14:53:12.765",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "192.0.3.33",
                 "hops.ip.max": 32},
  "results":    ["time",
                 "intermediate.ip4",
                 "hops.ip",
                 "delay.twoway.icmp.us"],
  "resultvalues": [ [ "2014-08-25 14:53:11.019", "192.0.2.1",       1, 162 ],
                    [ "2014-08-25 14:53:11.220", "217.147.223.101", 2, 15074 ],
```

```
                    [ "2014-08-25 14:53:11.570", "77.109.135.193",  3, 30093 ],
                    [ "2014-08-25 14:53:12.091", "77.109.135.34",   4, 34979 ],
                    [ "2014-08-25 14:53:12.310", "192.0.3.1",       5, 36120 ],
                    [ "2014-08-25 14:53:12.765", "192.0.3.33",      6, 36202 ]
                  ]

}
```

Indirect export to a repository with subsequent query requires three capabilities: one in which the repository advertises its ability to accept data over a given external protocol, one in which the probe advertises its ability to export data of the same type using that protocol, and one in which the repository advertises its ability to answer queries about the stored data. Returning to the aggregate ping measurement, first let's consider a repository which can accept these measurements via direct POST of mPlane result messages:

```
{
  "capability": "collect",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "ping-aggregate-collect",
  "when":       "past ... future",
  "export":     "mplane-https://repository.example.com:4343/result",
  "parameters": {"source.ip4":      "*",
                 "destination.ip4": "*"},
  "results":    ["delay.twoway.icmp.us.min",
                 "delay.twoway.icmp.us.mean",
                 "delay.twoway.icmp.us.50pct",
                 "delay.twoway.icmp.us.max",
                 "delay.twoway.icmp.count"]
}
```

This capability states that the repository at `https://repository.example.com:4343/result` will accept mPlane result messages matching the specified schema, without any limitations on time. Note that this schema matches that of the export capability provided by the probe:

```
{
  "capability": "measure",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "ping-aggregate-export",
  "when":       "now ... future / 1s",
  "export":     "mplane-https",
  "parameters": {"source.ip4":      "192.0.2.19",
                 "destination.ip4": "*"},
  "results":    ["delay.twoway.icmp.us.min",
                 "delay.twoway.icmp.us.mean",
```

```
                "delay.twoway.icmp.us.50pct",
                "delay.twoway.icmp.us.max",
                "delay.twoway.icmp.count"]
}
```

which differs only from the previous probe capability in that it states that results can be exported via the `mplane-https` protocol. Subsequent queries can be sent to the repository in response to the query capability:

```
{
  "capability": "query",
  "version":    0,
  "registry":   "http://ict-mplane.eu/registry/core",
  "label":      "ping-aggregate-query",
  "when":       "past ... future",
  "link":       "mplane-https://repository.example.com:4343/specification",
  "parameters": {"source.ip4":      "*",
                 "destination.ip4": "*"},
  "results":    ["delay.twoway.icmp.us.min",
                 "delay.twoway.icmp.us.mean",
                 "delay.twoway.icmp.us.50pct",
                 "delay.twoway.icmp.us.max",
                 "delay.twoway.icmp.count"]
}
```

3.2 mPlane over HTTPS

The default session protocol for mPlane messages is HTTP over TLS with mandatory mutual authentication. This grants confidentiality and integrity to the exchange of mPlane messages through a link security approach, and is transparent to the client. HTTP over TLS was chosen in part because of its ubiquitous implementation on many platforms.

An mPlane component may act either as a TLS server or a TLS client, depending on the workflow. When an mPlane client initiates a connection to a component, it acts as a TLS client, and must present a client certificate, which the component will verify against its allowable clients and map to an internal identity for making access control decisions before proceeding. The component, on the other hand, acts as a TLS server, and must present a server certificate, which the client will verify against its accepted certificates for the component before proceeding. When an mPlane component initiates a connection to a client (or, more commonly, the client interface of a supervisor), this arrangement is reversed: the component acts as a TLS client, the client as a TLS server, and mutual authentication is still mandatory. The mPlane client or component has an **identity** which is algorithmically derived from it's certificate's Distinguished Name (DN).

mPlane envisions a bidirectional message channel; however, unlike WebSockets and SSH described in the next subsections, HTTPS is not a bidirectional protocol. This makes it necessary to specify mappings between this bidirectional message channel and the sequence of HTTPS requests and

responses for each deployment scenario. These mappings are given in section 4. Note that in a given mPlane domain, any or all of these mappings may be used simultaneously.

When sending mPlane messages over HTTPS, the Content-Type of the message indicates the message representation. The MIME Content-Type for mPlane messages using JSON representation over HTTPS is `application/x-mplane+json`. When sending exceptions in HTTP response bodies, the response should contain an appropriate 400 (Client Error) or 500 (Server Error) response code. When sending indirections, the response should contain an appropriate 300 (Redirection) response code. Otherwise, the response should contain response code 200 OK.

3.2.1 mPlane PKI for HTTPS

The clients and components within an mPlane domain generally share a single certificate issuer, specific to a single mPlane domain. Issuing a certificate to a client or component then grants it membership within the domain. Any client or component within the domain can then communicate with components and clients within that domain. In a domain containing a supervisor, all clients and components within the domain can connect to the supervisor. This is necessary to scale mPlane domains to large numbers of clients and components without needing to specifically configure each client and component identity at the supervisor.

In the case of interdomain federation, where supervisors connect to each other, each supervisor will have its own issuer. In this case, each supervisor must be configured to trust each remote domain's issuer, but only to identify that domain's supervisor. This compartmentalization is necessary to keep one domain from authorizing components and clients within another domain.

3.2.2 Access control in HTTPS

For components with simple authorization policies (e.g., many probes), the ability to establish a connection implies verification of a client certificate valid within the domain, and further implies authorization to continue with any capability offered by the component. Conversely, in component-initiated workflows (see section 4.2), the ability of a component to connect to the supervisor implies that the supervisor will trust capabilities from that component.

For components with more complex policies (e.g., many repositories), an identity based on the DN of the peer's certificate is mapped to an internal identity on which access control decisions can be made. For access control purposes, the identity of an mPlane client or component is based on the Distinguished Name extracted from the certificate, which uniquely and securely identifies the entity carrying it.

In an mPlane domain containing a supervisor, each component trusts its supervisor completely, and accepts every message that can be identified as coming from the supervisor. Access control enforcement takes place on the supervisor, using a RBAC approach: an identity based on the DN extracted from their certificate of the clients is mapped to a role. Each role has access only to a subset of the whole set of capabilities provided by that to a supervisor, as composed from the capabilities offered by the associated components, according to its privileges. Therefore, any client will only have access to capabilities at the supervisor that it is authorized to execute. The same controls are enforced on specifications.

3.2.3 Paths in mPlane link and export URLs

In general, when connecting to a component for the first time, a client or supervisor will have been configured with a URL from which to retrieve capabilities. Conversely, when connecting to a client or supervisor for the first time, a component will have discovered or been configured with a URL to which to post capabilities. From there, every capability retrieved by a client should have a link section to which to POST specifications, and every specification retrieved by a component should have a link section to which to POST results.

However, in cases in which only an address (and not a full URL) is discoverable, given the ease of differentiating message handing in many web application frameworks by URL, mPlane HTTP clients and components can use the following convention: If a client can only discover a component's address, it should `GET /capabilities` to get that component's capabilities. If a client posts a specification for a capability that does not contain a link to a component, and only has that component's address, it should `POST` the specification to `/specification`. If a component wants to return results to a client and only has the client's address, and the corresponding specification does not have a link, it should `POST` the result to `/result`.

Additional path information can also be used in link and export section URLs to convey an implicit authorization from one component to another via a supervisor. Consider a repository which only wants to accept data from probes which a trusted supervisor has told to export to it. While the probes and repository share a domain by certificate issuer, the repository can further restrict access by placing a cryptographically random token in the export URL in the capability it gives to the supervisor e.g.:

`mplane-https://repository.example.com:4343/4e749ecb64d8dd6be986de03bebe/result`.

In this case, only components explicitly delegated by the supervisor can export to the repository. The same pattern can be used to delegate posting of specifications and results securely.

3.3 mPlane over WebSockets over TLS

Though not presently implemented by the SDK, the mPlane protocol specification is designed such that it can also use the WebSockets protocol as specified in RFC 6455 [5] as a session layer. Once an WebSockets connection is established, mPlane messages can be exchanged bidirectionally over the channel. A client may establish a connection to a component, or a component to a client, as required for a given application.

Access control in WebSockets is performed as in the HTTPS case: both clients and components are identified by certificates, identities derived from certificate DN, and domain membership is defined by certificate issuer.

Implementation and further specification of WebSockets as a session layer is a matter for future work. Though WebSockets is a better fit for the bidirectional nature of the mPlane protocol than HTTPS, the latter was chosen as mandatory to implement given the ubiquity of interoperable implementations of it for a diverse set of platforms. We suspect the situation with WebSockets will improve as implementations mature.

3.4 mPlane over SSH

Though not presently implemented by the SDK, the mPlane protocol specification is designed such that it can also use the Secure Shell (SSH) protocol as a session layer. In the SSH binding, a connection initiator (SSH client) identifies itself with an RSA, DSA, or ECDSA public key, which is bound to a specific identity, and the connection responder (SSH server) identifies itself with a host public key. Once an SSH connection is established, mPlane messages can be exchanged bidirectionally over the channel.

When using SSH as a session layer, clients and components are identified by SSH keys. SSH keys are not very human-readable identifiers, and as such must be mapped to identifiers at each component, client, and supervisor, on which roles can be assigned and access control decisions made. Additionally, SSH keys are not signed by an issuer, so there is no PKI-based definition of membership within a domain as with HTTPS. The need to specifically manage keys for every client and component, and the mappings to identities used in RBAC, will tend to limit the use of SSH to small domains.

Implementation and further specification of SSH as a session layer is a matter for future work. SSH was originally chosen as a possible session protocol for small domains, in order to save the overhead of building a PKI; however, implementation experience has shown that managing SSH keys manually has little administrative overhead advantage over using a small PKI with an algorithmic mapping from subject distinguished names to access control identities.

4 Workflows in HTTPS

As noted above, mPlane protocol supports three patterns of workflow: **client-initiated**, **component-initiated**, and **indirect export**. These workflow patterns can be combined into complex interactions among clients and components in an mPlane infrastructure. In the subsections below, we illustrate these workflows as they operate over HTTPS. Operation over WebSockets or SSH is much simpler: since the session protocol in these cases provides a bidirectional channel for message exchange, so the message sender and message exchange initiator are independent from the connection initiator, and callback control and capability discovery as described here are unnecessary.

In this figures in this section, the following symbols have the following meanings:

Symbol	Description
C	Capability
Ccb	Callback Capability
Ce	Export Capability
Cc	Collect Capability
S	Specification
Scb	Callback Specification
Se	Export Specification
R	Result
Rc	Receipt
Rd	Redemption
Ex	External protocol for indirect export
I	Interrupt

Colors are as elsewhere in the document: blue for capabilities and capability-related messages, red for specifications, and black for results.

4.1 Client-Initiated

Client-initiated workflows are appropriate for stationary components, i.e., those with stable, routable addresses, which can therefore act as HTTPS servers. This is generally the case for supervisors, large repositories, repositories acting as gateways to external data sources, and certain large-scale or public probes. The client-initiated pattern is illustrated in figure 4.1.

Here, the client opens an HTTPS connection the the component, and GETs a capability message, or an envelope containing capability messages, at a known URL. It then subsequently uses these capabilities by POSTing a specification, either to a known URL or to the URL given in the `link` section of the capability. The HTTP response to the POSTed specification contains either a result directly, or contains a receipt which can be redeemed later by POSTing a redemption to the component. This latter case is illustrated in figure 4.2.

In a client-initiated workflow with a delayed result, the client is responsible for polling the component with a redemption at the appropriate time. For measurements (i.e. specifications with the verb `measure'), this time is known as it is defined by the end of the temporal scope for the specification.

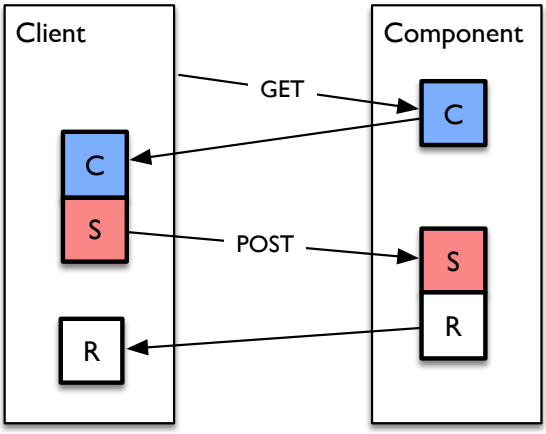

Figure 4.1: Client-initiated workflow

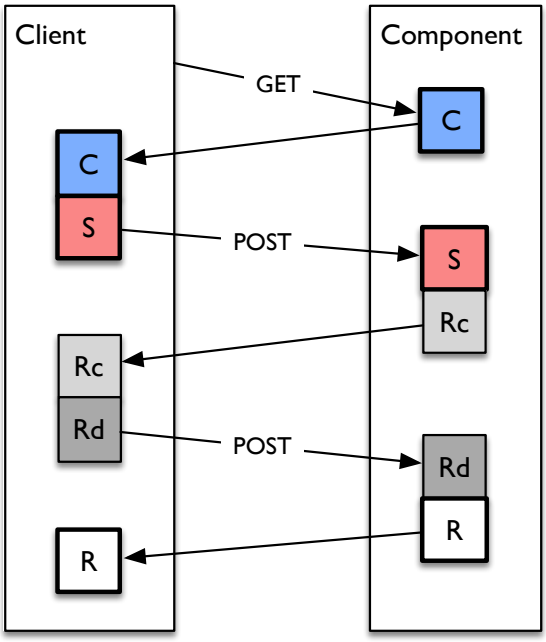

Figure 4.2: Client-initiated workflow with delayed result

Note that in client-initiated workflows, clients may store capabilities from components for later use: there may be a significant delay between retrieval of capabilities and transmission of specifications following from those capabilities. It is *not* necessary for a client to check to see whether a given capability it has previously retrieved is still valid before sending a specification.

4.1.1 Capability Discovery

For direct client-initiated workflows, the URL(s) from which to GET capabilities is a client configuration parameter. The client-initiated workflow also allows indirection in capability discovery. Instead of GETting capabilities direct from a component, they can also be retrieved from a *capability discovery server* containing capabilities for multiple components providing capabilities via client-initiated workflows. These components are then identified by the `link` section of each capability. The capabilities may be grouped in an envelope retrieved from the capability discovery server, or linked to in an HTML object retrieved therefrom.

In this way, a client needs only be configured with a single URL for capability discovery, instead of URLs for each component with which it wants to communicate. This arrangement is shown in figure 4.3.

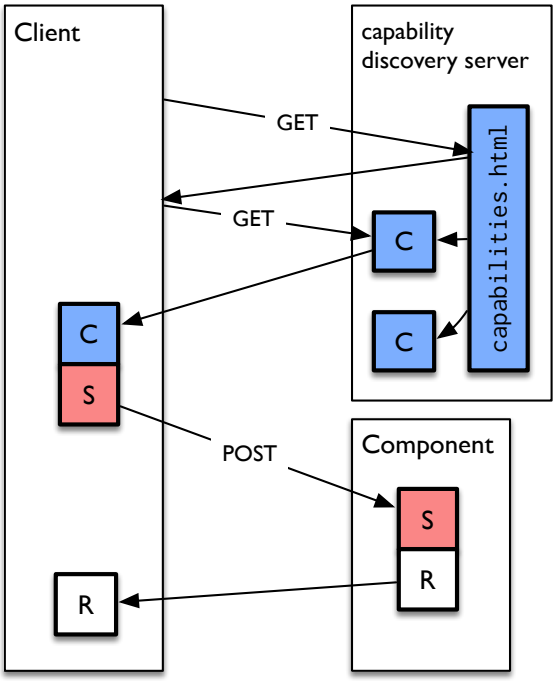

Figure 4.3: Capability discovery in client-initiated workflows

4.2 Component-Initiated

Component-initiated workflows are appropriate for components which do not have stable routable addresses (i.e., are behind NATs and/or are mobile), and which are used by clients that do. Common examples of such components are lightweight probes on mobile devices and customer equipment on access networks, interacting directly with a supervisor.

In this case, the usual client-server relationship is reversed, as shown in figure 4.4.

Here, when the component becomes available, it opens an HTTPS connection to the client and

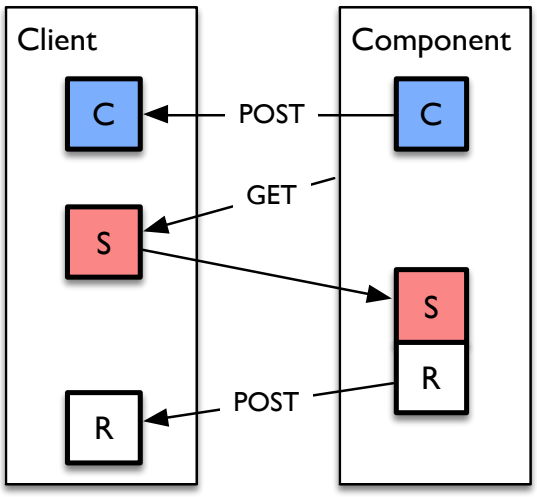

Figure 4.4: Component-initiated workflow

POSTs its capabilities to a known, configured URL at the supervisor. The supervisor remembers which capabilities it wishes to use on which components, and prepares specifications for later retrieval by the client.

The component then polls the supervisor, opening HTTPS connections and attempting to GET a specification from a known URL. The client will either respond 404 Not Found if the client has no current specification for the component, or with a specification to run matching a previously POSTed capability. After completing the measurement specified, the component then calls back and POSTs the results to the supervisor at a known URL.

In this case, the component must be configured with the client's URL(s).

4.2.1 Callback Control

Callback control allows the supervisor to specify to the component *when* it should call back, in order to allow centralized scheduling of component-initiated workflows, as well as to allow an mPlane infrastructure using component-initiated workflows to scale. Continuous polling of a client by thousands of components would put a network under significant load, and the polling delay introduces a difficult tradeoff between timeliness of specification and polling load. mPlane uses the `callback` verb with component-initiated workflows in order to allow the supervisor fine-grained control over when components will call back.

To use callback control, the component advertises the following capability along with the others it provides:

```
{
 'capability': 'callback',
 'version':    0,
 'registry':   'http://ict-mplane.eu/registry/core',
```

```
 'when':       'now ... future',
 'parameters': {},
 'results':    []
}
```

Then, when the component polls the client the first time, it responds with an envelope containing two specifications: the measurement it wants the client to perform, and a callback specification, containing the time at which the client should poll again in the temporal scope; e.g. as follows:

```
{
 'specification': 'callback',
 'version':       0,
 'registry':      'http://ict-mplane.eu/registry/core',
 'when':          '2014-09-08 12:40:00.000',
 'parameters':    {},
 'results':       []
}
```

Callback control is illustrated in figure 4.5.

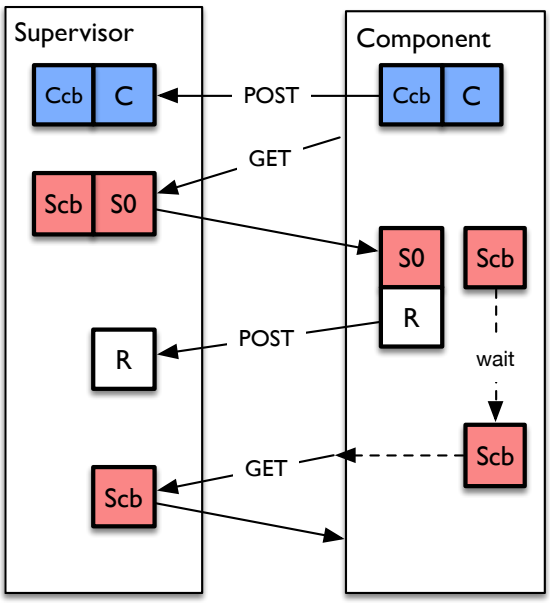

Figure 4.5: Callback control in component-initiated workflow

Note that if the supervisor has no work for the component, it returns a single callback specification as opposed to returning 404. Note that subsequent callback control specification to a component can have different time intervals, allowing a supervisor fine-grained control on a per-component basis of the tradeoff between polling load and response time.

Components implementing component-initiated workflows should support callback control in order to ensure the scalability of large mPlane infrastructures.

4.3 Indirect Export

Many common measurement infrastructures involve a large number of probes exporting large volumes of data to a (much) smaller number of repositories, where data is reduced and analyzed. Since (1) the mPlane protocol is not particularly well-suited to the bulk transfer of data and (2) fidelity is better ensured when minimizing translations between representations, the channel between the probes and the repositories is in this case external to mPlane. This **indirect export** channel runs either a standard export protocol such as IPFIX, or a proprietary protocol unique to the probe/repository pair. It coordinates an **exporter** which will produce and export data with a **collector** which will receive it. All that is necessary is that (1) the client, exporter, and collector agree on a schema to define the data to be transferred and (2) the exporter and collector share a common protocol for export.

An example arrangement is shown in figure 4.6.

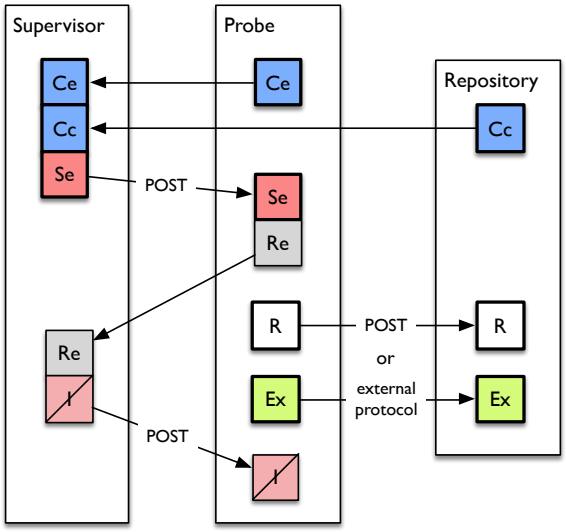

Figure 4.6: Indirect export workflow

Here, we consider a client speaking to an exporter and a collector. The client first receives an export capability from the exporter (with verb `measure` and with a protocol identified in the `export` section) and a collection capability from the collector (with the verb `collect` and with a URL in the `export` section describing where the exporter should export), either via a client-initiated workflow or a capability discovery server. The client then sends a specification to the exporter, which matches the schema and parameter constraints of both the export and collection capabilities, with the collector's URL in the `export` section.

The exporter initiates export to the collector using the specified protocol, and replies with a receipt that can be used to interrupt the export, should it have an indefinite temporal scope. In the meantime, it sends data matching the capability's schema directly to the collector.

This data, or data derived from the analysis thereof, can then be subsequently retrieved by a client using a client-initiated workflow to the collector.

4.4 Error Handling in mPlane Workflows

Any component may signal an error to its client or supervisor at any time by sending an exception message. While the taxonomy of error messages is at this time left up to each individual component, given the weakly imperative nature of the mPlane protocol, exceptions should be used sparingly, and only to notify components and clients of errors with the mPlane infrastructure itself.

It is generally presumed that diagnostic information about errors which may require external human intervention to correct will be logged at each component; the mPlane exception facility is not intended as a replacement for logging facilities (such as syslog).

Specifically, components in component-initiated workflows should not use the exception mechanism for common error conditions (e.g., device losing connectivity for small network-edge probes) -- specifications sent to such components are expected to be best-effort. Exceptions should also not be returned for specifications which would normally not be delayed but are due to high load -- receipts should be used in this case, instead. Likewise, specifications which cannot be fulfilled because they request the use of capabilities that were once available but are no longer should be answered with withdrawals.

Exceptions *should* always be sent in reply to messages sent to components or clients which cannot be handled due to a syntactic or semantic error in the message itself.

5 The Role of the Supervisor

From the point of view of the mPlane protocol, a supervisor is merely a combined component and client. The logic binding client and component interfaces within the supervisor is application-specific, as it involves the following operations according to the semantics of each application:

- translating lower-level capabilities from subordinate components into higher-level (composed) capabilities, according to the application's semantics
- translating higher-level specifications from subordinate components into lower-level (decomposed) specifications
- relaying or aggregating results from subordinate components to supervisor clients

The workflows on each side of the supervisor are independent; indeed, the supervisor itself will generally respond to client-initiated exchanges, and use both component-initiated and supervisor-initiated exchanges with subordinate components.

An example combination of workflows at a supervisor is shown in figure 5.1.

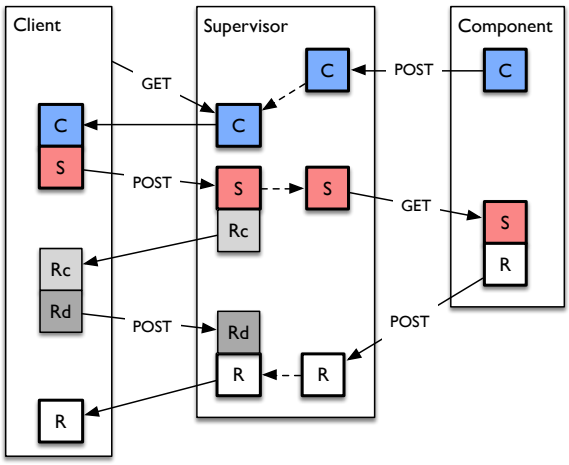

Figure 5.1: Example workflows at a supervisor

Here we see a a very simple arrangement with a single client using a single supervisor to perform measurements using a single component. The component uses a component-initiated workflow to associate with a supervisor, and the client uses a client-initiated workflow.

First, the component registers with the supervisor, POSTing its capabilities. The supervisor creates composed capabilities derived from these component capabilities, and makes them available to its client, which GETs them when it connects.

The client then initiates a measurement by POSTing a specification to the supervisor, which decomposes it into a more-specific specification to pass to the component, and hands the client a receipt for a the measurement. When the component polls the supervisor -- controlled, perhaps, by callback control as described above -- the supervisor passes this derived specification to the component, which executes it and POSTs its results back to the supervisor. When the client redeems its receipt, the supervisor returns results composed from those received from the component.

This simple example illustrates the three main responsibilities of the supervisor, which are described in more detail below.

5.1 Component Registration

In order to be able to use components to perform measurements, the supervisor must **register** the components associated with it. For client-initiated workflows -- large repositories and the address of the components is often a configuration parameter of the supervisor. Capabilities describing the available measurements and queries at large-scale components can even be part of the supervisor's externally managed static configuration, or can be dynamically retrieved and updated from the components or from a capability discovery server.

For component-initiated workflows, components connect to the supervisor and POST capabilities and withdrawals, which requires the supervisor to maintain a set of capabilities associated with a set of components currently part of the mPlane infrastructure it supervises.

5.2 Client Authentication

For many components -- probes and simple repositories -- very simple authentication often suffices, such that any client with a certificate with an issuer recognized as valid is acceptable, and all capabilities are available to. Larger repositories often need finer grained control, mapping specific peer certificates to identities internal to the repository's access control system (e.g. database users).

In an mPlane infrastructure, it is therefore the supervisor's responsability to map client identities to the set of capabilities each client is authorized to access. This mapping is part of the supervisor's configuration.

5.3 Capability Composition and Specification Decomposition

The most dominant responsibility of the supervisor is *composing* capabilities from its subordinate components into aggregate capabilities, and *decomposing* specifications from clients to more-specific specifications to pass to each component. This operation is always application-specific, as the semantics of the composition and decomposition operations depend on the capabilities available from the components, the granularity of the capabilities to be provided to the clients. It is for this reason that the mPlane SDK does not provide a generic supervisor.

6 Data Protection and Inter-Domain cooperation

6.1 Privacy and data protection

Within mPlane domains, the privacy of user data is protected by three techniques. First, all communication among components and clients is protected by transport level encryption (see section 3.2) ensuring confidentiality and integrity of the data transmitted over the mPlane protocol, with trust relationships managed by a closed public key infrastructure (PKI) (see section 3.2.1). Second, role-based access control at components ensures that data access is restricted to authorized entities(see section 3.2.2). Third, the principle of least access ensures that stored and measured data contain real identifying information (e.g., IP addresses) only when necessary for operations; otherwise, all identifiers handled within an mPlane domain are pseudonymized.

6.2 Access Control Model

The access control model has been implemented as described in section 3.2.2, and as elaborated below.

6.2.1 Authentication and Authorization

A mPlane component is authenticated using the certificate used in the TLS handshake and verified by each correspondent with the chain of trust of a certificate chain, that has been previously installed on each component. The distribution and the installation of all the root CA certificates needed for the communications must be performed out-of-band, because of the intrinsic lack of trust in the initial communication; i.e., a Time of First Use (TOFU) model has been deemed not secure enough.

The authorization security model has been implemented using a role-based (RBAC) model, where each permitted capability is mapped to one or more roles and each role is mapped to all those users that are authorized to use that capability. The RBAC authorization model has been chosen for the current reference implementation, because it permits a logical independence in specifying user authorizations, by decoupling the assignment of roles to users and the assignment of authorizations to access objects to roles. This greatly simplifies the overall authorization management. The authorization policy implemented is a closed policy, that allows an access to a capability only if exists an explicit authorization for it, and denies it otherwise. So when a client asks the supervisor for all exposed capabilities, only the authorized ones will be returned; in the same way when a specification is sent to the supervisor then a role-based authorization check is always performed to verify if that action is granted. The communication between probe and supervisor, as for the others components, is always performed using a TLS-based (HTTPS or WebSocket over TLS) channel. The identity of the supervisor is trusted if its certificate is successfully verified.

Small, simple probes often don't need fine-grained authorization (see section 5.2). That is, any of its capabilities are available to any successfully authenticated trusted entity within the domain, so often only the authentication of the supervisor is needed to establish a trusted channel.

In order to have a fine-grained authorization on the exposed capabilities, the client identification on the supervisor is performed extracting the Subject DN (Distinguished Name) from the X.509 certificate that the client has configured for establishing the HTTPS channel. This means that, while every component built using the SDK may perform access control based on the subject DN, in practice the RBAC authorization mechanism is configured only on the supervisor component, making the supervisor the center of trust of the domain.

The current implementation of the authorization management consists on a simple user-to-role and capability-to-roles mapping using plain-text configuration files. In the future, depending on requests of new features coming from the community that will grow around the mPlane project, different modules (e.g. LDAP and Active Directory support) should be developed and integrated for a much more scalable authorization management solution. For the same reason, at the current stage of development of the project, context-based constraints on access control (e.g. time-based access limits, maximum number of permitted operations, etc.) are not implemented in the SDK, even if the spatial one can be easily achieved with a classic firewall-based perimeter security approach.

6.2.2 PKI management

The overall mPlane PKI model and trust model is described in section 3.2.1. Usually the management of a root Certificate Authority or of different CAs (e.g. one for each mPlane partner) is a organizational issue that resides much more on a real-world production environment and it is definitely independent from the mPlane architecture. For testing and demonstration, some easy-to-use scripts based on the OpenSSL library are included with the mPlane SDK for the issue of client and server digital certificates. Using this ``PKI starter kit'', each partner can generate its own issuer CA (that belongs to the same mPlane root CA), that can be used to release all the certificates needed within the organization's domain and that will permit to all components to be automatically trusted within all the mPlane network.

6.2.3 Inter-domain communications

In mPlane the inter-domain communications are handled in the same way of other communications: since in this case there are two supervisors interacting with each other, each supervisor will map the DN of the external supervisor to an identity, and this identity to a role associated with certain privileges. Those privileges are established on the basis of agreements stipulated between the owners (or administrators) of the two domains. The only additional requirement of this kind of scenario could represented by the need of having a less granular trust relationship, for example using the certificate's issuer DN instead of the subject one (as explained in 6.3.2.2 of D1.2). The implementation of this feature, even if not currently available, should be easily achieved in the next releases of the Reference Implementation.

7 Implementations

There are two implementations of the mPlane protocol that are under development so far within the project. The use of two implementation, in two different languages, are especially valuable as they show interoperability of the mPlane protocol.

7.1 Reference Implementation and Software Development Kit

The reference implementation of the mPlane protocol is available on GitHub at `http://github.com/fp7-mplane/protocol-ri`. It is implemented in Python 3.3, additionally requiring the Tornado framework and urllib3 module. Documentation for the reference implementation API is automatically generated using Sphinx, and is available at `http://fp7mplane.github.io/protocol-ri/`.

As of the date of this deliverable, development is underway to build a Software Development Kit (SDK) based on the reference implementation, for use for demonstration software development within the project as well as to allow external developers to build mPlane-compatible software. This development can be found on the `sdk` branch of the repository, and will migrate to the `master` branch on release in the first half of 2015.

The SDK is composed of several modules:

- `mplane.model`: Information model and JSON representation of mPlane messages.
- `mplane.scheduler`: Component runtime scheduler. Maps capabilities to Python code that implements them (in `Service`) and keeps track of running specifications and associated results (`Job` and `MultiJob`).
- `mplane.tls`: Handles transport layer security for HTTPS, and maps local and peer certificates to identities for access control.
- `mplane.azn`: Handles access control, mapping identities to roles and authorizing roles to use specific Services.
- `mplane.client`: mPlane client framework. Handles client-initiated (`HttpClient`) and component-initiated (`ListenerHttpClient`) workflows.
- `mplane.clientshell`: Simple command-line shell for debugging of components and supervisors.
- `mplane.component`: mPlane component framework. Handles client-initiated (`ListenerHttpComponent`) and component-initiated (`InitiatorHttpComponent`) workflows.

7.2 NodeJS Implementation

In addition to the SDK, the project built a protocol implementation in NodeJS. More details about the libraries and the code can be found at `https://github.com/finvernizzi/mplane`. This second implementation was built first for ease of integration with Telecom Italia's DATI tool, as well as to

provide a second implementation to provide a basis for interoperability testing within the project, and the evaluation of the implementability of the protocol specification.

8 Initial Core Registry

The mPlane protocol is designed to have a flexible definition of its element registry, as described in section 2.1. The initial core registry for use during the project, identified by the url `http://www.ict-mplane.eu/registry/core`, was populated following an analysis of the use cases to be elaborated during the project in Deliverable 1.1. The elements in this registry are listed in the table below.

Name	Primitive	Desciption
start	time	Start time of an event/flow that may have a non-zero duration
end	time	End time of an event/flow that may have a non-zero duration
time	time	Time at which an single event occurred
duration.s	natural	Duration of an event/flow in seconds
duration.ms	natural	Duration of an event/flow in milliseconds
duration.us	natural	Duration of an event/flow in microseconds
duration.ns	natural	Duration of an event/flow in nanoseconds
source.ip4	address	Source IPv4 address of an event/flow, or the IPv4 address from which an active measurement was taken
source.ip6	address	Source IPv6 address of an event/flow, or the IPv6 address from which an active measurement was taken
source.port	natural	Source layer 4 port of an event/flow, or the port from which packets were sent when an active measurement was taken
source.interface	string	A locally-scoped identifier of an interface to which the source of an event/flow is attached, or from which an active measurement was taken
source.device	string	A locally-scoped identifier of a source device of an event/flow, or from which an active measurement was taken
source.as	natural	BGP AS number of the source of an event/flow, or AS originating an active measurement
destination.ip4	address	The destination IPv4 address of an event/flow, or the IPv4 address of the target of an active measurement
destination.ip6	address	The destination IPv6 address of an event/flow, or the IPv6 address of the target of an active measurement
destination.port	natural	The destination layer 4 port of an event/flow, or the port to which packets were sent when an active measurement was taken

Name	Primitive	Desciption
destination.interface	string	A locally-scoped identifier of an interface to which the destination of an event/flow is attached, or the target interface of an active measurement
destination.device	string	A locally-scoped identifier of a destination device of an event/flow, or the target of an active measurement
destination.as	natural	BGP AS number of the destination of an event/flow, or AS target of an active measurement
destination.url	url	A URL identifying a target of an active measurement
observer.ip4	address	The IPv4 address of the observation point of a passive measurement
observer.ip6	address	The IPv6 address of the observation point of a passive measurement
observer.link	string	A locally-scoped identifier of the link on which a passive measurement was observed
observer.interface	string	A locally-scoped identifier of the interface on which a passive measurement was observed
observer.device	string	A locally-scoped identifier of the device on which a passive measurement was observed
observer.as	natural	BGP AS number of the observer of a passive measurement or looking glass
intermediate.ip4	address	IPv4 address of a given entity along the path of a measurement; often scoped by hops.ip
intermediate.ip6	address	IPv6 address of a given entity along the path of a measurement; often scoped by hops.ip
intermediate.port	natural	Layer 4 port on which a flow/event was observed on a given entity along the path; used for NAPT applications
intermedate.as	natural	BGP AS number of a given entity along the path of a measurement; often scoped by hops.as
octets.ip	natural	Count of octets at layer 3 (including IP headers) associated with a flow, event, or measurement
octets.tcp	natural	Count of octets at layer 4 (including TCP headers) associated with a flow, event, or measurement
octets.udp	natural	Count of octets at layer 4 (including UDP headers) associated with a flow, event, or measurement

Name	Primitive	Desciption
octets.transport	natural	Count of octets at layer 4 (including all sub-network-layer headers) associated with a flow, event, or measurement
octets.layer5	natural	Count of octets at layer 5 (i.e., excluding network and transport layer headers) associated with a flow, event, or measurement
octets.layer7	natural	Count of octets at layer 7 (i.e., passed up to the application, excluding network and transport layer headers and octets in retransmitted packets) associated with a flow, event, or measurement
packets.ip	natural	Count of IP packets associated with flow, event, or measurement
packets.tcp	natural	Count of TCP segments associated with flow, event, or measurement
packets.udp	natural	Count of UDP segments associated with flow, event, or measurement
packets.transport	natural	Count of packets with a transport-layer header associated with a flow, event, or measurement
packets.layer5	natural	Count of packets with non-empty transport-layer payload associated with a flow, event, or measurement
packets.layer7	natural	Count of packets carrying unique data at layer 7 (i.e., packets.layer5 minus retransmissions) associated with a flow, event, or measurement
packets.duplicate	natural	Count of duplicated packets observed in a flow, event, or measurement
packets.outoforder	natural	Count of out-of-order packets observed in a flow, event, or measurement
packets.lost	natural	Count of packets observed or inferred as lost in a flow, event, or measurement
packets.unobserved	natural	Count of packets observed or inferred as delivered but unobserved in a flow, event, or measurement
flows	natural	Count of unidirectional flows (see RFC 7011) associated with an event or measurement
flows.bidirectional	natural	Count of bidirectional flows (see RFC 7011 and 5103) associated with an event or measurement
delay.twoway.icmp.us	natural	Singleton two-way delay in microseconds as measured by ICMP Echo Request/Reply (see RFC 792)

Name	Primitive	Desciption
delay.twoway.icmp.us.min	natural	Minimum two-way delay in microseconds as measured by ICMP Echo Request/Reply (see RFC 792)
delay.twoway.icmp.us.mean	natural	Mean two-way delay as in microseconds measured by ICMP Echo Request/Reply (see RFC 792)
delay.twoway.icmp.us.50pct	natural	Median two-way delay in microseconds as measured by ICMP Echo Request/Reply (see RFC 792)
delay.twoway.icmp.us.max	natural	Maximum two-way delay in microseconds as measured by ICMP Echo Request/Reply (see RFC 792)
delay.twoway.icmp.count	natural	Count of valid ICMP Echo Replies received when measuring two-way delay using ICMP Echo Request/Reply (see RFC 792)
delay.oneway.owamp.us	natural	Singleton one-way delay along a path as measured by OWAMP (see RFC 3763) in microseconds
delay.oneway.owamp.us.min	natural	Minimum one-way delay along a path as measured by OWAMP (see RFC 3763) in microseconds
delay.oneway.owamp.us.mean	natural	Mean one-way delay along a path as measured by OWAMP (see RFC 3763) in microseconds
delay.oneway.owamp.us.50pct	natural	Median one-way delay along a path as measured by OWAMP (see RFC 3763) in microseconds
delay.oneway.owamp.us.max	natural	Maximum one-way delay along a path as measured by OWAMP (see RFC 3763) in microseconds
delay.oneway.owamp.count	natural	Count of samples for one-way delay measurements using OWAMP (see RFC 3763)
delay.queue.us	natural	Singleton measured or inferred delay attributable to queueing along a path in microseconds
delay.queue.us.min	natural	Minimum measured or inferred delay attributable to queueing along a path in microseconds
delay.queue.us.mean	natural	Mean measured or inferred delay attributable to queueing along a path in microseconds
delay.queue.us.50pct	natural	Median measured or inferred delay attributable to queueing along a path in microseconds
delay.queue.us.max	natural	Maximum measured or inferred delay attributable to queueing along a path in microseconds
delay.buffer.us	natural	Delay attributable to buffering at an endpoint in microseconds
delay.resolution.ms	natural	Delay from transaction start to completion of resolution of a name or URL to an address, in milliseconds

Name	Primitive	Desciption
delay.firstbyte.ms	natural	Delay from transaction start to receipt of first byte of content at the initiator, in milliseconds
rtt.ms	natural	Round-trip time as measured or estimated at the sender in milliseconds
rtt.us	natural	Round-trip time as measured or estimated at the sender in microseconds
iat.ms	natural	Packet interarrival or event interoccurance time in milliseconds
iat.us	natural	Packet interarrival or event interoccurance time in microseconds
connectivity.ip	boolean	Assertion (or negation) that layer 3 connectivity between the identified source and destination is available
connectivity.as	boolean	Assertion (or negation) that control plane connectivity (i.e. BGP routability) between the identified source and destination is available
hops.ip	natural	Count of layer 3 hops or subhops along the identified path
hops.as	natural	Count of control-plane hops or subhops along the identified path
bandwidth.nominal.bps	natural	Nominal (advertised) bandwidth at a point or along a path in bits per second
bandwidth.nominal.kbps	natural	Nominal (advertised) bandwidth at a point or along a path in kilobits per second
bandwidth.nominal.Mbps	natural	Nominal (advertised) bandwidth at a point or along a path in megabits per second
bandwidth.partial.bps	natural	Partial bandwidth attributable to a given flow in bits per second
bandwidth.partial.kbps	natural	Partial bandwidth attributable to a given flow in kilobits per second
bandwidth.partial.Mbps	natural	Partial bandwidth attributable to a given flow in megabits per second
bandwidth.imputed.bps	natural	Bandwidth assumed to be available along a path according to measurement and heuristics in bits per second
bandwidth.imputed.kbps	natural	Bandwidth assumed to be available along a path according to measurement and heuristics in kilobits per second

Name	Primitive	Desciption
bandwidth.imputed.Mbps	natural	Bandwidth assumed to be available along a path according to measurement and heuristics in megabits per second
content.url	url	A URL identifying some content, access to which is passively or actively measured
fps.nominal	float	Nominal frame rate in frames per second of the identified audio/video content
fps.achieved	float	Achieved frame rate in frames per second of the identified audio/video content
fps.achieved.min	float	Minumum achieved frame rate in frames per second of the identified audio/video content
fps.achieved.mean	float	Mean achieved frame rate in frames per second of the identified audio/video content
fps.achieved.max	float	Maximum achieved frame rate in frames per second of the identified audio/video content
sessions.transport	natural	Count of transport-layer sessions associated with an event
sessions.layer7	natural	Count of application-layer sessions associated with an event
cpuload	real	Normalized CPU load on the identified device
memload	real	Normalized memory load on the identified device
linkload	real	Normalized link load on the identified interface or link
bufferload	real	Normalized buffer load on the identified device
bufferstalls	natural	Count of buffer stalls (imputed playback quality degradation) associated with a flow/event
snr	real	Signal to noise ratio in decibels, either in a radio access network or in an audio transmission context
measurement.identifier	string	Free-form string identifying the implementation of the measurement on the component; often the name of the external program
measurement.revision	natural	Release or deployment serial number of the implementation of the measurement on the component
measurement.algorithm	string	Free-form string identifying the algorithm used for the measurement on the component
location.latitude	float	The latitude of the component expressed as a floating point number of degrees north of the equator

Name	Primitive	Desciption
location.longitude	float	The longitude of the component expressed as a floating point number of degrees east of the standard meridian (on Earth, the Prime Meridian at Greenwich)
location.altitude	float	The altitude of the component expressed as a floating point number of meters above the standard zero altitude (on Earth, mean sea level)
location.civil	string	A free-form identifier of the civil location (postal address, city name, building name, etc) of the component

References

[1] B. Trammell, ed. Use Case Elaboration and Requirements Specification. mPlane Public Deliverable 1.1.

[2] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, Mar. 2014. (Proposed Standard).

[3] B. Claise, B. Trammell, and P. Aitken. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Flow Information. RFC 7011 (Internet Standard), Sept. 2013.

[4] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008. (Proposed Standard) Updated by RFCs 5746, 5878, 6176.

[5] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, Dec. 2011. (Proposed Standard).

[6] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, June 2014. (Proposed Standard).

[7] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291, Feb. 2006. (Proposed Standard) Updated by RFCs 5952, 6052, 7136, 7346, 7371.

[8] S. Kawamura and M. Kawashima. A Recommendation for IPv6 Address Text Representation. RFC 5952, Aug. 2010. (Proposed Standard).

[9] G. Klyne and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339, July 2002. (Proposed Standard).

[10] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[11] B. Trammell. Textual Representation of IP Flow Information Export (IPFIX) Abstract Data Types. RFC 7373, Sept. 2014. (Proposed Standard).